



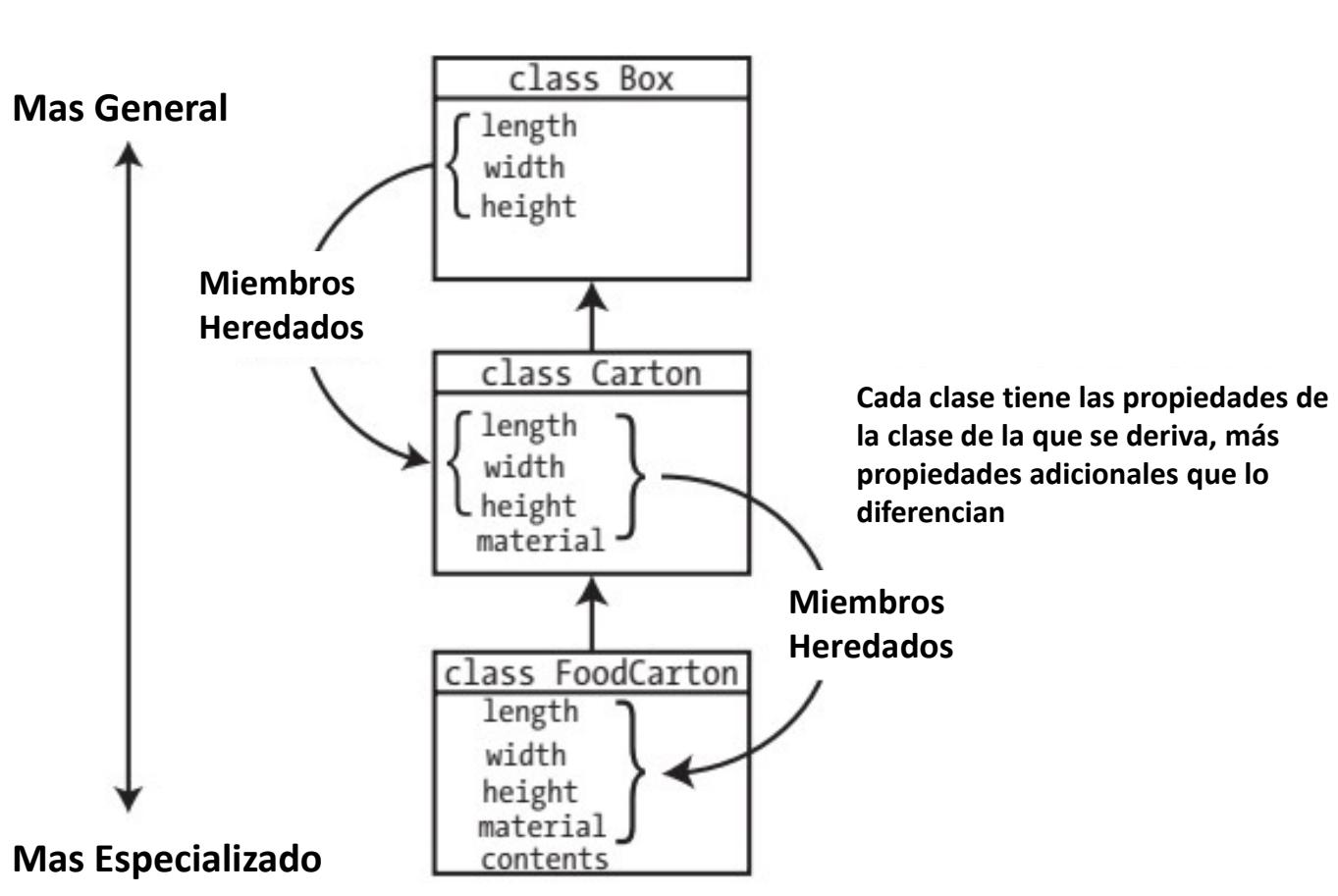
C++ SEMANA 3

HERENCIA

Podemos crear tipos especializados de objetos a partir de uno general. En nuestro caso podríamos crear objetos Box especializados. Por ejemplo, una clase Carton podría tener las mismas propiedades que un objeto Box, es decir, las tres dimensiones, además de la propiedad adicional de su composición material (de qué tipo de cartón está hecho).

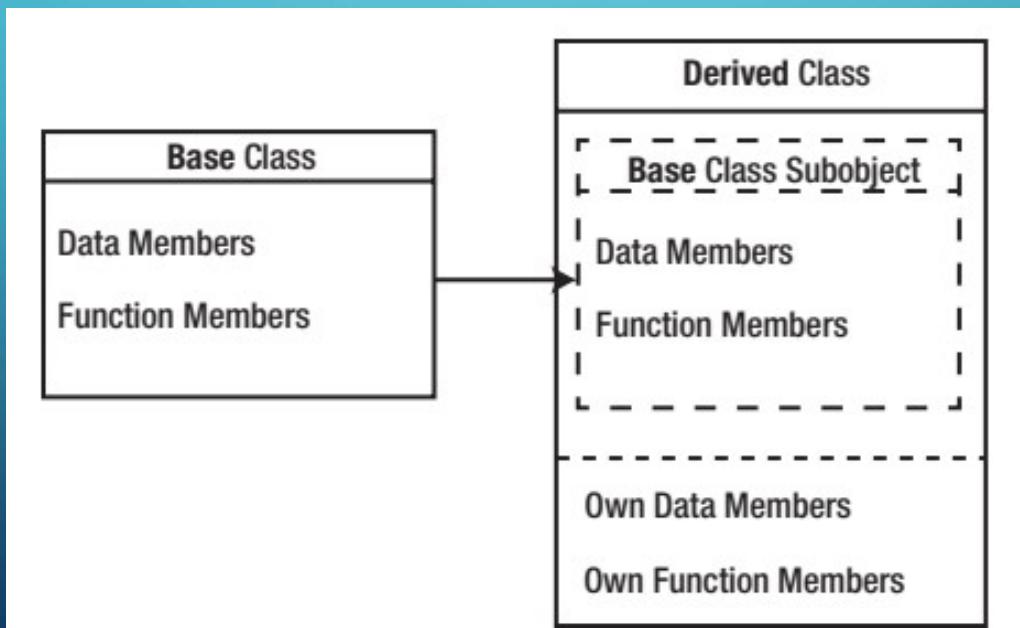
Luego, podría especializarse aún más utilizando la definición Carton para describir una clase FoodCarton, que es un tipo especial de Carton que está diseñado para contener alimentos. Un objeto FoodCarton tendrá todas las propiedades de un objeto Carton y un miembro adicional para modelar los contenidos. Por supuesto, un objeto Carton tiene las propiedades de un objeto Box, por lo que un objeto FoodCarton también las tendrá.

HERENCIA



HERENCIA

La clase Carton es una clase base directa de FoodCarton. Debido a que Carton se define en términos de la clase Box, la clase Box es una clase base indirecta de la clase FoodCarton. Un objeto de la clase FoodCarton tendrá miembros heredados de Carton, incluidos los miembros que la clase Carton hereda de la clase Box.



HERENCIA

Derivando clases

Podemos ver a continuación como derivamos clases en C++. El siguiente ejemplo muestra la clase Box y Carton. Cabe destacar que hemos utilizado la sobrecarga de operadores para sobre cargar el operador << del flujo de salida. Esto nos permite imprimir una representación en string del objeto en cuestión (Box, Carton, etc).

Podemos definir una clase Carton basada en la clase Box. Un objeto Carton será similar a un objeto Box pero con una variable de miembro adicional que indica el material del que está hecho. Definiremos Carton como una clase derivada, utilizando la clase Box como clase base.

HERENCIA

Ejemplo 1:

```
#ifndef BOX_H_ // impide la inclusión recursiva
#define BOX_H_

#include <iostream>
#include <iomanip>

class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructores
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {}
    Box() = default; // Constructor sin argumentos.

    double volume() const { return length*width*height; }

    // getters
    double getLength() const { return length; }
    double getWidth() const { return width; }
    double getHeight() const { return height; }
};
```

HERENCIA

Continuación de definición de clase Box.

```
// toString C++ sobrecarga del operador <<
inline std::ostream& operator<<(std::ostream& stream, const Box& box)
{
    stream << " Box(" << std::setw(2) << box.getLength() << ','
    << std::setw(2) << box.getWidth() << ','
    << std::setw(2) << box.getHeight() << ')';
    return stream;
}
#endif

#endif /* BOX_H_ */
```

HERENCIA

Clase Carton, derivada de clase Box.

```
#ifndef CARTON_H_
#define CARTON_H_

#include <string>
#include <string_view>
#include "Box.h"

class Carton : public Box
{
private:
    std::string material;

public:
    explicit Carton(std::string mat = "Cardboard") : material{mat} {} // Constructor
};

#endif /* CARTON_H_ */
```

HERENCIA

La directiva `#include` para la definición de la clase `Box` es necesaria porque es la clase base para `Carton`. La primera línea de la definición de la clase `Carton` indica que `Carton` se deriva de `Box`. El nombre de la clase base sigue a dos puntos que lo separan del nombre de la clase derivada, `Carton` en este caso. La palabra clave `public` es un especificador de acceso de clase base que determina cómo se puede acceder a los miembros de `Box` desde dentro de la clase `Carton`.

En todos los demás aspectos, la definición de la clase `Carton` se parece a cualquier otra. Contiene un nuevo miembro, `material`, que es inicializado por el constructor. El constructor define un valor predeterminado para la cadena que describe el material de un objeto `Carton` para que este también sea el constructor sin argumentos para la clase `Carton`.

HERENCIA

Los objetos Carton contienen todas las variables miembro de la clase base, Box, además de la variable miembro adicional, material. Debido a que heredan todas las características de un objeto Box, los objetos Carton también son objetos Box. Hay una insuficiencia evidente en la clase Carton en el sentido de que no tiene un constructor definido que permita establecer los valores de los miembros heredados, pero volveremos a eso más adelante. Veamos cómo funcionan estas definiciones de clase en un ejemplo:

HERENCIA

```
#include <iostream>
#include "Box.h"
#include "Carton.h"

int main() {

    // Instanciar un objeto Box y dos objetos Carton.
    Box box {40.0, 30.0, 20.0};
    Carton carton;
    Carton chocolateCarton {"Cartón sólido blanqueado"};

    // Ocupación de memoria de los objetos Base y Derivados.
    std::cout << "box ocupa " << sizeof box << " bytes" << std::endl;
    std::cout << "carton ocupa " << sizeof carton << " bytes" << std::endl;
    std::cout << "chocolateCarton ocupa: " << sizeof chocolateCarton << " bytes" << std::endl;

    // Calculo de volumenes
    std::cout << "Volumen de box: " << box.volume() << std::endl;
    std::cout << "Volumen de carton: " << carton.volume() << std::endl;
    std::cout << "Volumen de chocolateCarton: " << chocolateCarton.volume() << std::endl;
    std::cout << "Largo de chocolateCarton: " << chocolateCarton.getLength() << std::endl;

    return 0;
}
```

HERENCIA

La función main() crea un objeto Box y dos objetos Carton y genera el número de bytes ocupados por cada objeto. El resultado muestra lo que cabría esperar: que un objeto Carton es más grande que un objeto Box. Un objeto Box tiene tres variables miembro de tipo double; cada uno de estos ocupa 8 bytes en casi todas las máquinas, por lo que son 24 bytes en total. Ambos objetos Carton tienen el mismo tamaño: 56 bytes. La memoria adicional ocupada por cada objeto Carton se reduce al material de la variable miembro, por lo que es del tamaño de un objeto string que contiene la descripción del material. La salida de los volúmenes para los objetos Carton muestra que la función volume() se hereda de hecho en la clase Carton y que las dimensiones tienen los valores predeterminados de 1.0. La siguiente declaración muestra que las funciones de acceso también se heredan y se pueden llamar para un objeto de clase derivado.

HERENCIA

Ahora, intente agregar la siguiente línea dentro de los miembros públicos de la clase Carton.

```
double carton_volume() const { return length*width*height; }
```

Esto no compilará. La razón es que aunque las variables miembro de Box se heredan, se heredan como miembros privados de la clase Box. El especificador de acceso **private** determina que los miembros son totalmente privados para la clase. No solo no se puede acceder a ellos desde fuera de la clase Box, sino que tampoco se puede acceder desde dentro de una clase que los herede.

El acceso a los miembros heredados de un objeto de clase derivada no solo está determinado por su especificación de acceso en la clase base, sino también por el especificador de acceso en la clase base y el especificador de acceso de la clase base en la clase derivada. Vamos a entrar en eso un poco más a continuación.

HERENCIA

A menudo, desea que los miembros de una clase base sean accesibles desde dentro de la clase derivada pero que, sin embargo, estén protegidos de interferencias externas. Además de los especificadores de acceso público y privado para miembros de clase, puede declarar miembros como protegidos (*protected*). Dentro de la clase, la palabra clave *protected* tiene el mismo efecto que la palabra clave *private*. No se puede acceder a los miembros protegidos desde fuera de la clase, excepto desde las funciones que se han especificado como funciones amigas. Sin embargo, las cosas cambian en una clase derivada. Los miembros de una clase base que se declaran como protegidos son de libre acceso en las funciones miembro de una clase derivada, mientras que los miembros privados de la clase base no lo son. Podemos modificar la clase Box para tener variables miembro protegidas:

HERENCIA

```
class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
...
// Resto de las definiciones de clase
...
};
```

Ahora, los campos de Box siguen siendo privados en el sentido de que no se puede acceder a ellos mediante funciones globales ordinarias, pero ahora se puede acceder a ellas dentro de las funciones miembro de una clase derivada. Si ahora intenta compilar Carton con el miembro carton_volume() incorporado y los miembros de la clase Box especificados como protegidos, encontrará que se compila sin problemas.

HERENCIA

Los campos normalmente siempre deben ser privados. el ejemplo anterior fue solo para indicar lo que es posible hacer. En general, las campos protegidos presentan problemas similares a los de las variables de miembros públicos, solo que en menor medida.

HERENCIA

En la definición de la clase Carton, especificamos la clase base Box como pública usando la siguiente sintaxis: `class Carton : public Box`. En general, existen tres posibilidades para el especificador de acceso de clase base: público, protegido o privado. Si omite el especificador de acceso de clase base en una definición de clase, el valor predeterminado es privado (en una definición de estructura, el valor predeterminado es público). Por ejemplo, si omite el especificador por completo escribiendo `class Carton : Box` en la parte superior de la definición de la clase Carton, entonces se asume el especificador de acceso privado para Box. Ya sabe que los especificadores de acceso para los miembros de la clase también vienen en tres tipos.

HERENCIA

Una vez más, la elección es la misma: pública, protegida o privada. El especificador de acceso de clase base afecta el estado de acceso de los miembros heredados en una clase derivada. Hay nueve combinaciones posibles. Cubriremos todas las combinaciones posibles en los siguientes párrafos, aunque la utilidad de algunas de ellas solo se hará evidente veamos polimorfismo.

Primero, consideremos cómo se heredan los miembros privados de una clase base en una clase derivada. Independientemente del especificador de acceso de la clase base (público, protegido o privado), un miembro de la clase base privada siempre permanece privado para la clase base.

HERENCIA

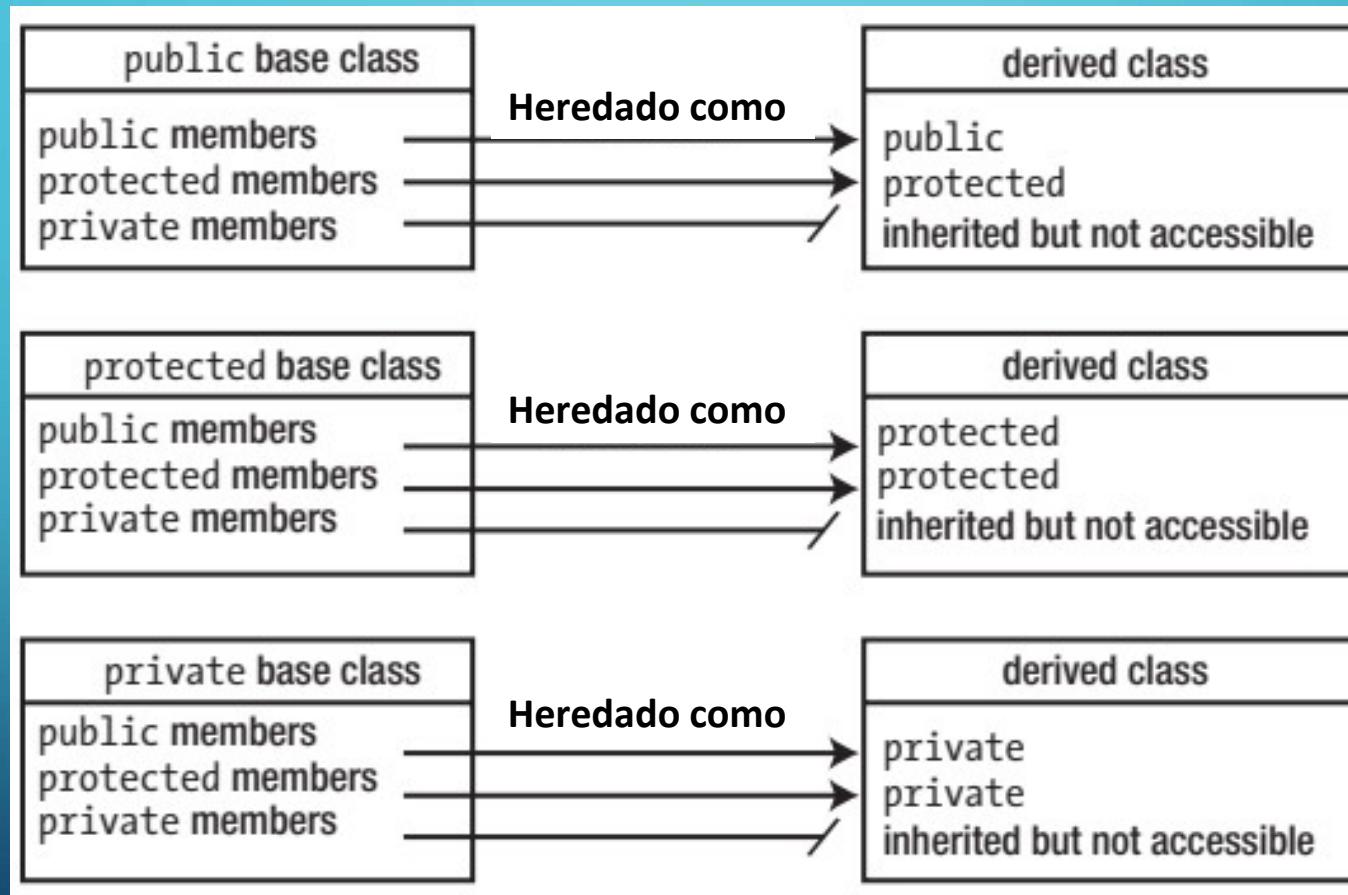
Control de acceso en clases derivadas

Como ha visto, los miembros privados heredados son miembros privados de la clase derivada, por lo que no se puede acceder a ellos fuera de la clase derivada. También son inaccesibles para las funciones miembro de la clase derivada porque son privadas para la clase base. Ahora, veamos cómo se heredan los miembros de la clase base públicos y protegidos. En todos los casos restantes, las funciones miembro de la clase derivada pueden acceder a los miembros heredados. La herencia de los miembros de la clase base públicos y protegidos funciona así:

HERENCIA

1. Cuando el especificador de clase base es público, el estado de acceso de los miembros heredados permanece sin cambios. Por lo tanto, los miembros públicos heredados son públicos y los miembros protegidos heredados están protegidos en una clase derivada.
2. Cuando el especificador de clase base está protegido, los miembros públicos y protegidos de una clase base se heredan como miembros protegidos.
3. Cuando el especificador de clase base es privado, los miembros públicos y protegidos heredados se vuelven privados para la clase derivada, por lo que las funciones miembro de la clase derivada pueden acceder a ellos, pero no se puede acceder a ellos si se heredan en otra clase derivada.

HERENCIA



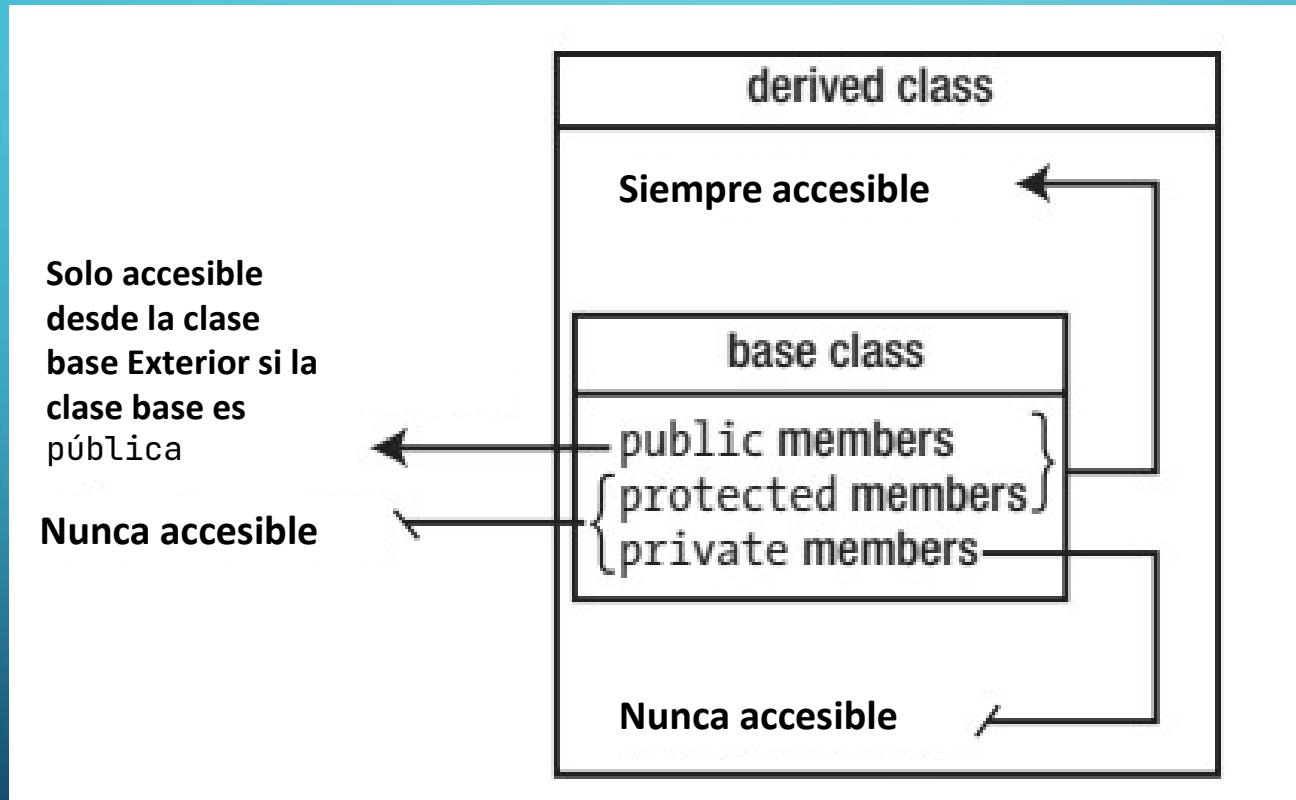
HERENCIA

Especificadores de acceso y jerarquías de clases

En la siguiente imagen se muestra cómo la accesibilidad de los miembros heredados se ve afectada únicamente por los especificadores de acceso de los miembros de la clase base. Dentro de una clase derivada, los miembros de la clase base públicos y protegidos siempre están accesibles, y los miembros de la clase base privada nunca están accesibles. Desde fuera de la clase derivada, solo se puede acceder a los miembros de la clase base pública, y este es el caso solo cuando la clase base se declara como pública.

HERENCIA

Especificadores de acceso y jerarquías de clases



HERENCIA

Especificadores de acceso y jerarquías de clases

Si el especificador de acceso de clase base es público, el estado de acceso de los miembros heredados permanece sin cambios. Al usar los especificadores de acceso de clase base privada y protegida, puede hacer dos cosas:

- Puede evitar el acceso a los miembros de la clase base pública desde fuera de la clase derivada; cualquiera de los especificadores lo hará. Si la clase base tiene funciones de miembros públicos, entonces este es un paso serio porque la interfaz de clase para la clase base se elimina de la vista pública en la clase derivada.
- Puede afectar cómo los miembros heredados de la clase derivada se heredan en otra clase que usa la clase derivada como base.

HERENCIA

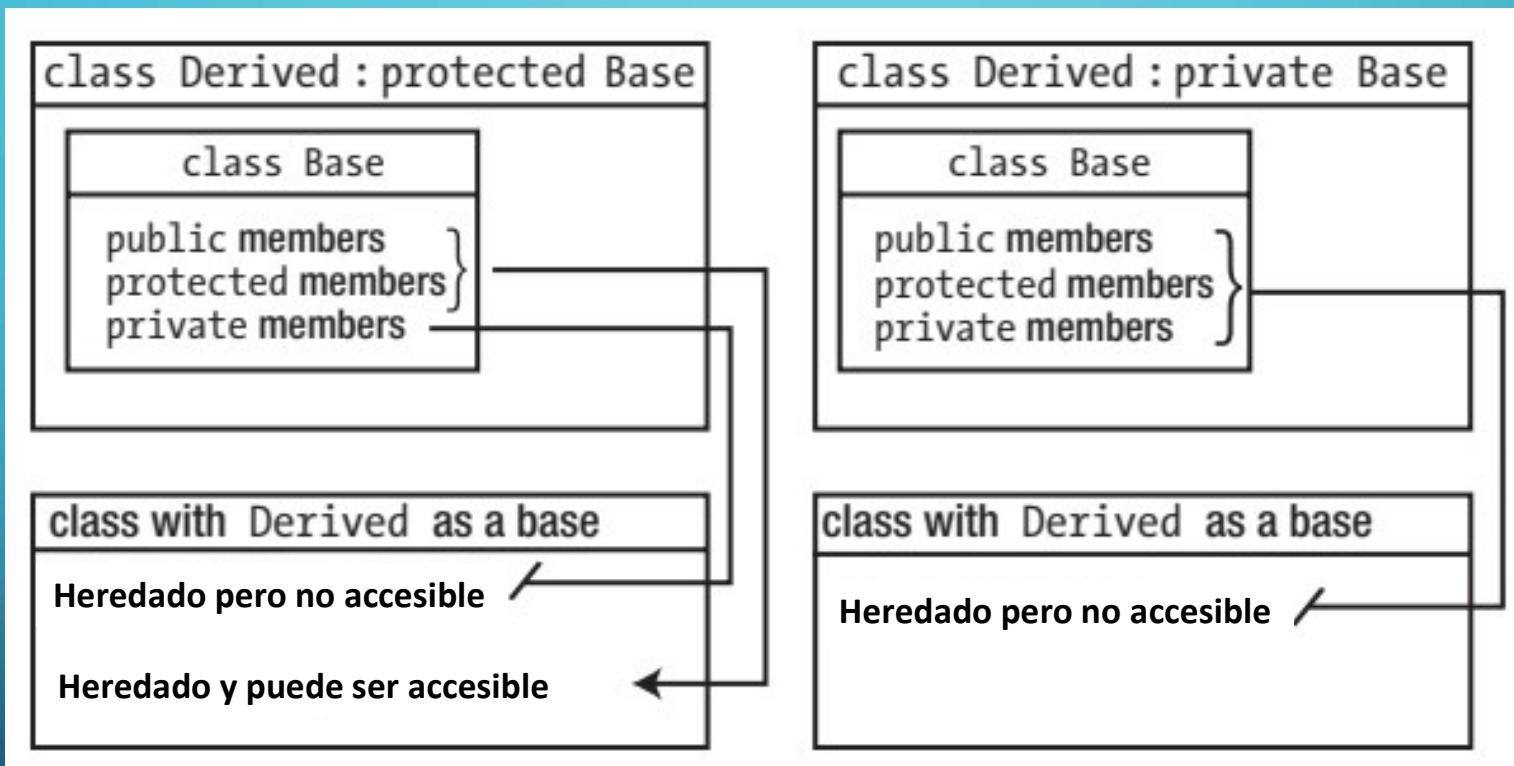
Especificadores de acceso y jerarquías de clases

La siguiente imagen muestra cómo los miembros públicos y protegidos de una clase base se pueden pasar como miembros protegidos de otra clase derivada. Los miembros de una clase base heredada de forma privada no serán accesibles en ninguna otra clase derivada. En la mayoría de los casos, es más apropiado aplicar el especificador de acceso de clase base como `public`, con los campos de la clase base declaradas como privadas o protegidas.

En este caso, las partes internas del subobjeto de la clase base son internas del objeto de la clase derivada y, por lo tanto, no forman parte de la interfaz pública del objeto de la clase derivada. En la práctica, debido a que el objeto de la clase derivada es un objeto de la clase base, deseará que la interfaz de la clase base se herede en la clase derivada, y esto implica que la clase base debe especificarse como `public`.

HERENCIA

Especificadores de acceso y jerarquías de clases



Los constructores normalmente no se heredan por muy buenas razones, pero verá más adelante en este capítulo cómo puede hacer que los constructores se hereden en una clase derivada.

HERENCIA

Elección de especificadores de acceso en jerarquías de clases

Debe tener en cuenta dos aspectos al definir una jerarquía de clases: los especificadores de acceso para los miembros de cada clase y el especificador de acceso de la clase base en cada clase derivada. Los miembros públicos de una clase definen la interfaz externa de la clase, y esto normalmente no debería incluir ninguna variable miembro.

SUGERENCIA *Como regla general, los campos de una clase siempre deben ser privados. Si el código fuera de la clase requiere acceso a las variables miembro, debe agregar métodos getters y setters públicos o protegidos.*

HERENCIA

Elección de especificadores de acceso en jerarquías de clases

Siempre recuerde que debe favorecer el ocultamiento de los campos de una clase:

- La ocultación de datos le permite preservar la integridad del estado de un objeto.
- Reduce el acoplamiento y las dependencias con código externo, facilitando así la evolución y los cambios ya sea en la representación interna de una clase o en la implementación concreta de sus funciones de interfaz.

HERENCIA

Elección de especificadores de acceso en jerarquías de clases

- Le permite inyectar código extra para ser ejecutado por cada acceso y/o modificación de variables miembro. Más allá de las comprobaciones de validez y chequeo de sanidad, esto puede incluir código de logging y depuración, por ejemplo, o mecanismos de notificación de cambios.
- Facilita la depuración, ya que la mayoría de los entornos de desarrollo admiten la colocación de los llamados puntos de interrupción de depuración en las llamadas a funciones. Poner puntos de interrupción en getters y setters hace que sea mucho más fácil rastrear qué código lee o escribe en las variables miembro y cuando.

HERENCIA

Elección de especificadores de acceso en jerarquías de clases

A menudo se olvida es que los campos protegidos tienen muchas de las mismas desventajas que los públicos:

- No hay nada que impida que una clase derivada invalide el estado de un objeto, lo que puede, por ejemplo, invalidar las llamadas invariantes de clase (propiedades del estado de un objeto que deberían mantenerse en todo momento) con las que cuenta el código en la clase base.
- Una vez que las clases derivadas manipulan directamente las variables miembro de una clase base, cambiar su implementación interna se vuelve imposible sin cambiar también todas las clases derivadas.
- Cualquier código adicional agregado a los métodos público getters y setters en la clase base se anula si las clases derivadas pueden omitirlo.

HERENCIA

Elección de especificadores de acceso en jerarquías de clases

- La interrupción de una sesión de depuración cuando se modifican las variables miembro se vuelve, al menos, más difícil si las clases derivadas pueden acceder a ellas directamente, siendo imposible la interrupción cuando se lean.

Por lo tanto, siempre haga que los campos sean privados, a menos que tenga una buena razón para no hacerlo.

Por cuestiones de simplicidad usaremos en algunos ejemplos miembros `protected`, esto debería evitarse en código profesional.

HERENCIA

Elección de especificadores de acceso en jerarquías de clases

Los métodos que no forman parte de la interfaz pública de una clase tampoco deben ser accesibles directamente desde fuera de la clase, lo que significa que deben ser privados o protegidos. La especificación de acceso que elija para una función en particular depende de si desea permitir el acceso desde dentro de una clase derivada.

HERENCIA

Cambio de la especificación de acceso de los miembros heredados

Es posible que desee eximir a un miembro de clase base en particular de los efectos de una especificación de acceso de clase base protegida o privada. Esto es más fácil de entender con un ejemplo. Suponga que deriva la clase Carton pero con Box como clase base privada. Todos los miembros heredados de Box ahora serán privados en Carton, pero le gustaría que la función de volumen () permanezca pública en la clase derivada, como lo es en la clase base. Puede restaurar el estado público de un miembro heredado en particular que era público en la clase base con una declaración using.

Esto es esencialmente lo mismo que la declaración using para espacios de nombres. Puede forzar que la función volume() sea pública en la clase derivada definiendo la clase Carton de esta manera:

HERENCIA

Cambio de la especificación de acceso de los miembros heredados

```
class Carton : private Box
{
private:
    std::string material;

public:
    using Box::volume; // Heredado como público
    explicit Carton(std::string mat = "Cardboard") : material {mat} {} // Constructor
};
```

La especificación de acceso a miembros se aplica a la declaración de uso, por lo que el nombre `volume` se introduce en la sección pública de la clase `Carton` para que anule la especificación de acceso de la clase base privada para el miembro `volume()` de la clase base. La función se heredará como pública en la clase `Cartón`, no como privada.

HERENCIA

Cambio de la especificación de acceso de los miembros heredados

Hay varios puntos a tener en cuenta aquí. Primero, cuando aplica una declaración de uso al nombre de un miembro de una clase base, debe calificar el nombre con el nombre de la clase base, porque esto especifica el contexto para el nombre del miembro. En segundo lugar, tenga en cuenta que no proporciona una lista de parámetros ni un tipo de devolución para una función miembro, solo el nombre calificado. Esto implica que *las funciones sobrecargadas siempre vienen como un paquete*. En tercer lugar, la declaración de uso también funciona con variables miembro heredadas en una clase derivada.

Puede usar una declaración `using` para anular un especificador de acceso público o privado original de la clase base.

HERENCIA

Cambio de la especificación de acceso de los miembros heredados

Por ejemplo, si la función `volume()` estaba protegida en la clase base `Box`, podría hacerla pública en la clase `Carton` derivada con la misma declaración de uso en una sección pública de `Carton`. Sin embargo, no puede aplicar una declaración de uso para relajar la especificación de un miembro privado de una clase base porque no se puede acceder a los miembros privados en una clase derivada.

HERENCIA

Constructores en una clase derivada

Los objetos de clase derivados siempre se crean de la misma manera, incluso cuando hay varios niveles de derivación. Primero se llama al constructor de la clase base seguido del constructor de la clase derivada de esta, seguido del constructor de la clase derivada de esta, y así sucesivamente, hasta que se llama al constructor de la clase más derivada. Un objeto de clase derivado tiene un objeto de clase base completo en su interior, y este debe crearse antes que el resto del objeto de clase derivado. Si esa clase base se deriva de otra clase, se aplica lo mismo.

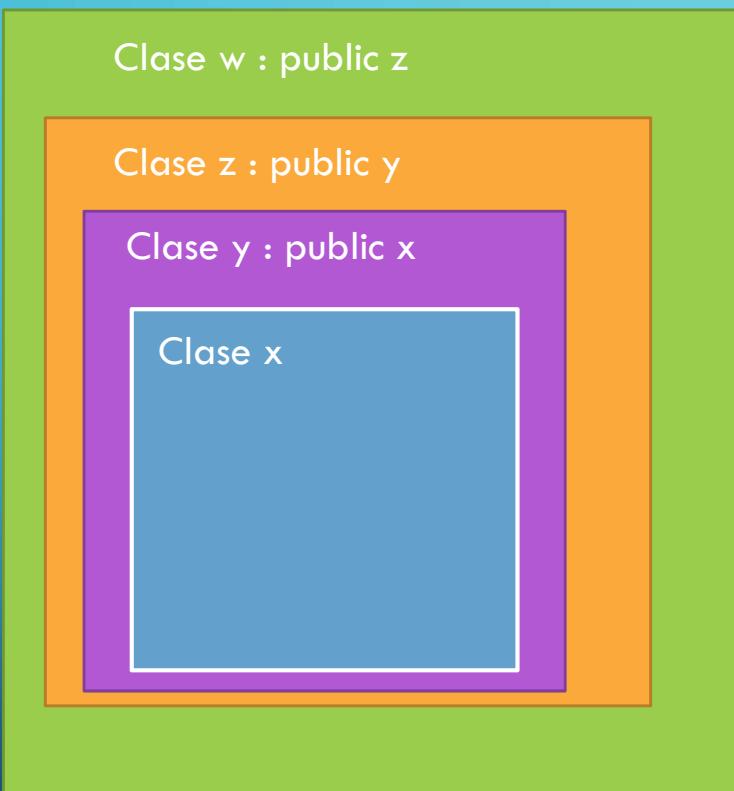
HERENCIA

Constructores en una clase derivada

Aunque en el **Ejemplo 1** se invocó automáticamente al constructor de clase base predeterminado, este no tiene por qué ser el caso. Puede llamar a un constructor de clase base particular en la lista de inicialización del constructor de clase derivada. Esto le permitirá inicializar las variables miembro de la clase base con un constructor que no sea el predeterminado. También le permitirá elegir un constructor de clase base en particular, según los datos proporcionados al constructor de la clase derivada. Veámoslo funcionando en otro ejemplo. Diríjase al repositorio, y descargue la carpeta Clase9_Ejemplo 2 que se encuentra en la carpeta Semana3.

HERENCIA

Constructores en una clase derivada – Orden de llamada



Orden de construcción

Clase x
Clase y
Clase z
Clase w

HERENCIA

Constructores en una clase derivada

Aunque se puede acceder a las variables miembro heredadas que no son privadas para la clase base desde una clase derivada, no se pueden inicializar en la lista de inicialización para un constructor de clase derivada. Por ejemplo, intente reemplazar el primer constructor de la clase Carton en el ejemplo anterior con lo siguiente:

```
Carton::Carton(double lv, double wv, double hv, std::string mat)
: length{lv}, width{wv}, height{hv}, material{mat}
{ std::cout << "Carton(double,double,double,string) llamado.\n"; }
```

HERENCIA

Constructores en una clase derivada

Como se ha dado cuenta no compila satisfactoriamente. Puede esperar que esto funcione porque la longitud, el ancho y la altura son miembros de la clase base protegidos (**protected**) que se heredan públicamente, por lo que el constructor de la clase Carton debería poder acceder a ellos. Sin embargo, el compilador se queja de que la longitud, el ancho y la altura no son miembros de la clase clase Carton. Este será el caso incluso si hace públicas las variables miembro de la clase Box. Si desea inicializar explícitamente las variables miembro heredadas, puede hacerlo en el cuerpo del constructor de la clase derivada. La siguiente definición de constructor funcionaría:

HERENCIA

Constructores en una clase derivada

```
// Este constructor si compila.  
Carton::Carton(double lv, double wv, double hv, std::string mat) : material{mat}  
{  
    length = lv;  
    width = wv;  
    height = hv;  
    std::cout << "Carton(double,double,double,string) llamado.\n";  
}
```

En el momento en que el cuerpo del constructor Carton comienza a ejecutarse, se ha creado la parte base del objeto. En este caso, la parte base del objeto Carton se crea mediante una llamada implícita del constructor de la clase Box sin argumentos. Posteriormente, puede hacer referencia a los nombres de los miembros de la clase base no privada sin ningún problema. Aún así, si es posible, *siempre es mejor reenviar los argumentos del constructor a un constructor de clase base apropiado y hacer que la clase base se encargue de inicializar los miembros heredados.*

HERENCIA

El constructor de copia en una clase derivada

Ya sabe que se llama al constructor de copias cuando se crea un objeto y se inicializa con otro objeto del mismo tipo de clase. El compilador proporcionará un constructor de copia predeterminado que crea el nuevo objeto copiando el objeto original miembro por miembro si no ha definido su propia versión. Ahora examinemos el constructor de copias en una clase derivada. Para hacer esto, agregaremos un constructor de copias a la clase base, Box, insertando el siguiente código en la sección pública de la definición de la clase.:

```
// Constructor de copia
Box(const Box& box) : length{box.length}, width{box.width},
height{box.height}
{ std::cout << "Constructor de copia de Box" << std::endl; }
```

HERENCIA

El constructor de copia en una clase derivada

Intentemos una primera versión del constructor de copia de Carton:

```
// Constructor de copia de Carton
Carton(const Carton& carton) : material {carton.material}
{ std::cout << "Constructor de copia de Carton" << std::endl; }
```

Probemos nuestro primer intento:

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

int main()
{
    // Declara e inicializa un objeto Carton
    Carton carton(20.0, 30.0, 40.0, "Papel glassine");
    std::cout << std::endl;

    Carton cartonCopy(carton); // Invoca al constructor de copia
    std::cout << std::endl;
    std::cout << "Volumen de carton: " << carton.volume() << std::endl
        << "Volumen de cartonCopy: " << cartonCopy.volume() << std::endl;
}
```

HERENCIA

El constructor de copia en una clase derivada

Salida de nuestro programa:

```
Box(double, double, double) llamado.  
Carton(double,double,double,string) llamado.
```

```
Box() llamado.  
Constructor de copia de Carton
```

```
Volumen de carton: 24000  
Volumen de cartonCopy: 1
```

HERENCIA

El constructor de copia en una clase derivada

Claramente, el volumen de `cartonCopy` no es el mismo que el de `carton`, pero el resultado también muestra la razón de esto. Para copiar el objeto `carton`, llama al constructor de copia para la clase `Carton`. El constructor de copias de `Carton` debe hacer una copia del subobjeto `Box` de `carton`, y para hacer esto debe llamar al constructor de copias de `Box`. Sin embargo, el resultado muestra claramente que en su lugar se está llamando al constructor de `Box` predeterminado (sin argumentos). **El constructor de copias de `Carton` no llamará al constructor de copias de `Box` si no le indica que lo haga.** El compilador sabe que tiene que crear un subobjeto `Box` para la objeto `carton`, pero si no especifica cómo, el simplemente creará un objeto base predeterminado.:

HERENCIA

El constructor de copia en una clase derivada

Cuando define un constructor para una clase derivada, es responsable de garantizar que los miembros del objeto de la clase derivada se inicialicen correctamente. Esto incluye todas las variables miembro heredadas directamente, así como las variables miembro que son específicas de la clase derivada. Además, esto se aplica a cualquier constructor, incluidos los constructores de copias.

Si no se especifica explícitamente un constructor de la clase derivada invocará al constructor por defecto de la clase base.

HERENCIA

El constructor de copia en una clase derivada

Para solucionar nuestro problema debemos corregir el constructor de copia de la clase Carton de la siguiente manera:

```
Carton(const Carton& carton) : Box{carton}, material{carton.material}  
{ std::cout << "Constructor de copia de Carton" << std::endl; }
```

El constructor de copia de Box se llama con el objeto carton como argumento. El objeto carton es del tipo Carton, pero también es un objeto Caja perfectamente bueno. El parámetro para el constructor de copia de la clase Box es una referencia a un objeto Box, por lo que el compilador pasará carton como tipo Box&, lo que dará como resultado que solo la parte base del cartón se pase al constructor de copia de Box. Modifique el constructor de copia de Carton y observe los resultados.

HERENCIA

El constructor por defecto en una clase derivada

Para solucionar nuestro problema debemos corregir el constructor de copia de la clase Carton de la siguiente manera:

```
Carton(const Carton& carton) : Box{carton}, material{carton.material}  
{ std::cout << "Constructor de copia de Carton" << std::endl; }
```

El constructor de copia de Box se llama con el objeto carton como argumento. El objeto carton es del tipo Carton, pero también es un objeto Caja perfectamente bueno. El parámetro para el constructor de copia de la clase Box es una referencia a un objeto Box, por lo que el compilador pasará carton como tipo Box&, lo que dará como resultado que solo la parte base del cartón se pase al constructor de copia de Box. Modifique el constructor de copia de Carton y observe los resultados.

HERENCIA

El constructor por defecto en una clase derivada

Sabe que el compilador no proporcionará un constructor predeterminado sin argumentos si define uno o más constructores para una clase. También sabe que puede decirle al compilador que inserte un constructor predeterminado en cualquier evento usando la palabra clave `predeterminada`. Podría reemplazar la definición del constructor sin argumentos en la definición de la clase `Carton` con esta declaración:

```
Carton() = default;
```

HERENCIA

El constructor por defecto en una clase derivada

Ahora el compilador proporcionará una definición, aunque haya definido otros constructores. La definición que proporciona el compilador para una clase derivada llama al constructor de la clase base, por lo que se ve así:

```
Carton() : Box{} {};
```

Esto implica que si el compilador proporciona el constructor sin argumentos en una clase derivada, debe existir un constructor sin argumentos no privado en la clase base.

HERENCIA

El constructor por defecto en una clase derivada

Si no es así, el código no se compilará. Puede demostrar esto fácilmente eliminando el constructor sin argumentos de la clase Box en el ejemplo Clase9_Ejemplo2 o haciéndolo privado. Con el constructor predeterminado proporcionado por el compilador especificado para la clase Carton, el código ya no se compilará. Cada constructor de clase derivada llama a un constructor de clase base. Si un constructor de clase derivada no llama explícitamente a un constructor base en su lista de inicialización, se llamará al constructor sin argumentos.

HERENCIA

Heredando constructores

Los constructores de clase base normalmente no se heredan en una clase derivada. Esto se debe a que una clase derivada normalmente tiene variables miembro adicionales que deben inicializarse, y un constructor de clase base no las conocería. Sin embargo, puede hacer que los constructores se hereden de una clase base directa colocando una declaración de uso en la clase derivada. Así es como se podría hacer una versión de la clase `Carton` de Clase9_Ejemplo2 para heredar los constructores de la clase `Box`:

```
class Carton : public Box
{
    using Box::Box; // Hereda los constructores de la clase Box
private:
    std::string material {"Cardboard"};

public:
    Carton(double lv, double wv, double hv, std::string_view mat)
        : Box{lv, wv, hv}, material{mat}
    { std::cout << "Carton(double,double,double,string_view) called.\n"; }
};
```

HERENCIA

Heredando constructores

Si la definición de la clase Box es la misma que en Clase9_Ejemplo2, la clase Carton heredará dos constructores: Box(double, double, double) y Box(double). Los constructores en la clase derivada se verán así:

```
Carton(double lv, double wv, double hv) : Box {lv, wv, hv} {}
explicit Carton(double side) : Box{side} {}
```

Cada constructor heredado tiene la misma lista de parámetros que el constructor base y llama al constructor base en su lista de inicialización. El cuerpo de cada constructor está vacío. Puede agregar más constructores a una clase derivada que hereda de su base directa, como ilustra el ejemplo de la clase Carton.

HERENCIA

Heredando constructores

A diferencia de los métodos regulares, los constructores (no privados) se heredan utilizando el mismo especificador de acceso que el constructor correspondiente en la clase base. Entonces, aunque la declaración de `Box::Box` es parte de la sección implícitamente privada de la clase `Carton`, los constructores heredados de `Box` son públicos. Si la clase `Box` hubiera tenido constructores protegidos, estos también se habrían heredado como constructores protegidos en `Carton`. Observe que un constructor de `Box` falta en la lista de constructores heredados: el constructor predeterminado. Es decir, la declaración de uso no hizo que se heredara un constructor predeterminado de la siguiente forma:

```
Carton() : Box{} {}
```

HERENCIA

Heredando constructores

Los constructores predeterminados nunca se heredan. Y debido a que Carton define explícitamente un constructor (por cierto, los constructores heredados no cuentan aquí), el compilador tampoco generó un constructor predeterminado. Técnicamente hablando, los constructores de copia tampoco se heredan, pero no lo notará, ya que el compilador genera principalmente un constructor de copia predeterminado de todos modos. Puede probar esto modificando Clase9_Ejemplo2, comentando el constructor sin argumentos de la clase Carton y creando los objetos mostrados en el siguiente fragmento en main():

HERENCIA

Heredando constructores

```
class Carton : public Box
{
private:
    std::string material {"Cardboard"};

public:
    Carton(double lv, double wv, double hv, std::string mat) : Box{lv, wv, hv}, material{mat}
    { std::cout << "Carton(double,double,double,string) llamado.\n"; }

    explicit Carton(std::string mat) : material{mat}
    { std::cout << "Carton(string) llamado.\n"; }

    Carton(double side, std::string_view mat) : Box{side}, material{mat}
    { std::cout << "Carton(double,string) llamado.\n"; }

    //Carton() { std::cout << "Carton() llamado.\n"; } // Constructor sin argumentos comentado.

    // Constructor de copia de Carton
    Carton(const Carton& carton) : material {carton.material}
    { std::cout << "Constructor de copia de Carton" << std::endl; }

};
```

HERENCIA

Heredando constructores

```
// Carton cart; // No compila: el constructor por defecto no se hereda
Carton cube{4.0}; // Llama al constructor heredado
Carton cartcopy { cube }; // Llama al constructor de copia por defecto
Carton carton {1.0, 2.0, 3.0}; // Llama al constructor heredado
Carton candyCarton (50.0, 30.0, 20.0, "Cartón fino"); // Llama al constructor de la
// clase Carton
```

HERENCIA

Destructores bajo herencia

La destrucción de un objeto de clase derivada involucra tanto al destructor de clase derivada como al destructor de clase base. Puede demostrar esto agregando destructores con declaraciones de salida en las definiciones de clase Box y Carton. Agregue la definición del destructor a la clase Box y el destructor de la clase Carton:

```
// Destructor Box
~Box() {
    std::cout << "Destructor Box" << std::endl;
}
```

```
// Destructor Carton
~Carton()
{
    std::cout << "Destructor carton. Material = " << material << std::endl;
}
```

HERENCIA

Destructores bajo herencia

Por supuesto, si las clases asignaron memoria de almacenamiento libre y almacenaron la dirección en un puntero crudo, sería esencial definir el destructor de clase para evitar pérdidas de memoria. El destructor de carton imprime el material para que pueda saber qué objeto de cartón se está destruyendo asignando un material diferente a cada uno. Veamos cómo se comportan estas clases en la práctica:

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

int main()
{
    Carton carton;
    Carton candyCarton{50.0, 30.0, 20.0, "Carton fino"};
    std::cout << "Volumen de carton:" << carton.volume() << std::endl;
    std::cout << "Volumen de candyCarton: " << candyCarton.volume() << std::endl;
}
```

HERENCIA

Destructores bajo herencia

Por supuesto, si las clases asignaron memoria de almacenamiento libre y almacenaron la dirección en un puntero crudo, sería esencial definir el destructor de clase para evitar pérdidas de memoria. El destructor de carton imprime el material para que pueda saber qué objeto de cartón se está destruyendo asignando un material diferente a cada uno. Veamos cómo se comportan estas clases en la práctica:

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

int main()
{
    Carton carton;
    Carton candyCarton{50.0, 30.0, 20.0, "Carton fino"};
    std::cout << "Volumen de carton:" << carton.volume() << std::endl;
    std::cout << "Volumen de candyCarton: " << candyCarton.volume() << std::endl;
}
```

HERENCIA

Destructores bajo herencia

El objetivo de este ejercicio es ver cómo se comportan los destructores. La salida de las llamadas al destructor indica dos aspectos de cómo se destruyen los objetos. En primer lugar, puede ver el orden en que se invocan los destructores para un objeto en particular y, en segundo lugar, puede ver el orden en que se destruyen los objetos. Las llamadas al destructor registradas por la salida corresponden a las siguientes acciones:

Salida Destructor	Objeto destruido
Destructor Carton. Material: Carton fino	Objeto candyCarton
Destructor Box	Subobjeto box de candyCarton
Destructor Carton. Material: Cardboard	Objeto carton
Destructor Box	Subobjeto box de carton

HERENCIA

Destructores bajo herencia

Esto muestra que los objetos que componen un objeto de clase derivado se destruyen en el orden inverso al que se crearon. El objeto carton se creó primero y se destruyó al final; el objeto candyCarton se creó en último lugar y se destruyó primero. Este orden se elige para garantizar que nunca termine con un objeto en un estado ilegal. Un objeto se puede usar solo después de que se haya definido; esto significa que cualquier objeto dado solo puede contener punteros (o referencias) que apuntan (o se refieren) a objetos que ya se han creado. Destruyendo un objeto dado antes que cualquier objeto al que pueda apuntar (o hacer referencia), se asegura de que la ejecución de un destructor no pueda dar como resultado punteros o referencias no válidos.

HERENCIA

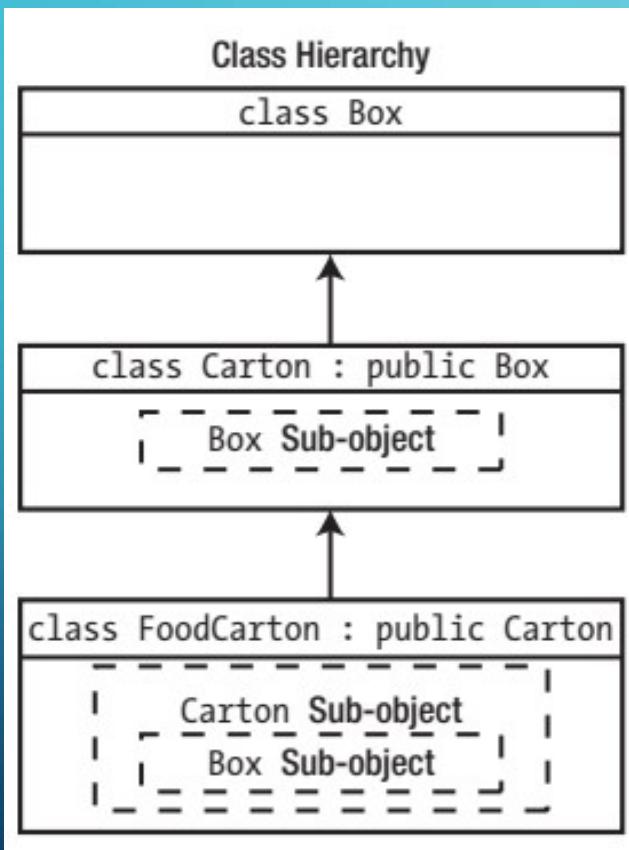
El orden en el cual los destructores son llamados

El orden de las llamadas al destructor para un objeto de clase derivado es el inverso de la secuencia de llamadas al constructor para el objeto. Primero se llama al destructor de clases derivadas y luego se llama al destructor de clases base, tal como en el ejemplo.

Para un objeto con varios niveles de clases de derivación, este orden de llamadas de destructor recorre la jerarquía de clases, comenzando con el destructor de clase más derivado y terminando con el destructor de la clase más básica.

HERENCIA

El orden en el cual los deestructores son llamados



Secuencia de constructores para crear un objeto FoodCarton:

1. **Constructor Box**
2. **Constructor Carton**
3. **Constructor FoodCarton**

Secuencia de deestructores para destruir un objeto FoodCarton:

1. **Destructor FoodCarton**
2. **Destructor Carton**
3. **Destructor Box**

HERENCIA

Nombres de campos duplicados

Es posible que una clase base y una clase derivada tengan cada una un campo con el mismo nombre. Si tiene mucha mala suerte, es posible que incluso tenga nombres duplicados en la clase base y en una base indirecta. Por supuesto, esto es confuso, y nunca debe proponerse deliberadamente crear un arreglo de este tipo en sus propias clases. Sin embargo, las circunstancias o los descuidos pueden hacer que así resulten las cosas. Entonces, ¿qué sucede si las variables miembro en las clases base y derivada tienen los mismos nombres?

La duplicación de nombres no es un impedimento para la herencia, y puede diferenciar entre miembros de clases base y derivados con nombres idénticos. Suponga que tiene una clase Base, definida de la siguiente manera:

HERENCIA

Nombres de campos duplicados

```
class Base
{
public:
    Base(int number = 10) : value{number} {} // Constructor

protected:
    int value;
};
```

Esto solo contiene una sola variable miembro (`value`) y un constructor.

Puede derivar una clase derivada de `Base` de la siguiente manera:

```
class Derived : public Base
{
public:
    Derived(int number = 20) : value{number} {} // Constructor
    int total() const; // Valor total de campos

protected:
    int value;
};
```

HERENCIA

Nombres de campos duplicados

La clase derivada tiene un campo llamado `value` y también heredará el miembro `value` de la clase base. Le mostraremos cómo puede distinguir los dos miembros con nombre `value` en la clase derivada escribiendo una definición para la función `total()`. Dentro del método de la clase derivada, `value` por sí mismo se refiere al miembro declarado dentro de ese ámbito, es decir, el miembro de la clase derivada. El miembro de la clase base se declara dentro de un ámbito diferente y, para acceder a él desde una método de clase derivada, debe calificar el nombre del miembro con el nombre de la clase base. Por lo tanto, puede escribir la función `total()` de la siguiente manera:

```
int Derived::total() const
{
    return value + Base::value;
}
```

La expresión `Base::value` se refiere al miembro de la clase base, y `value` por sí mismo se refiere al miembro declarado en la clase derivada.

HERENCIA

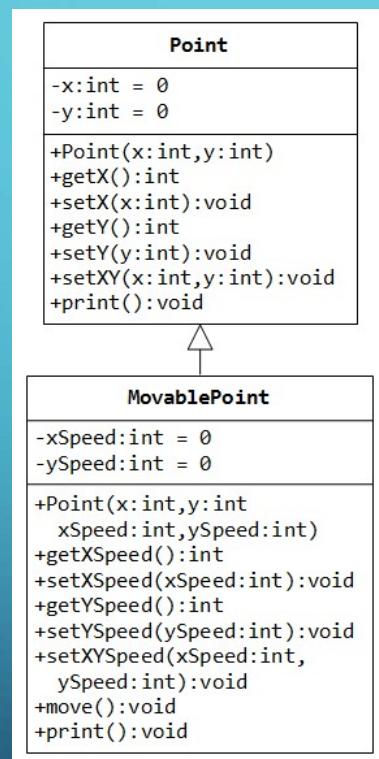
Ejercicio: Defina una clase base llamada Animal que contenga dos miembros campos privados : una cadena para almacenar el nombre del animal (por ejemplo, "Fido" o "Yogi") y un miembro entero llamado peso que contendrá el peso del animal en kilos. También incluya un método público, quien(), que genera un mensaje con el nombre y el peso del objeto Animal. Derive dos clases llamadas Leon y Cerdo Hormiguero, con Animal como clase base pública. Escriba una función main() para crear objetos Leon y Cerdo hormiguero("Leon" en 180 kilos y "Cerdo Hormiguero" en 30 kilos, digamos) y demuestre que el miembro quien() se hereda en ambas clases derivadas llamándolo para la clase derivada.

HERENCIA

Ejercicio: Cree un nuevo tipo de empaque que derive de la clase Carton llamado PackRigido, dicho empaque está destinado a productos con un embalaje especial, por tanto el volumen útil (en cuanto al producto contenido) es del 75% del volumen total de la caja. Implemente los métodos constructores y el método volumen para este empaque. El material de este empaque es cartón rígido.

HERENCIA

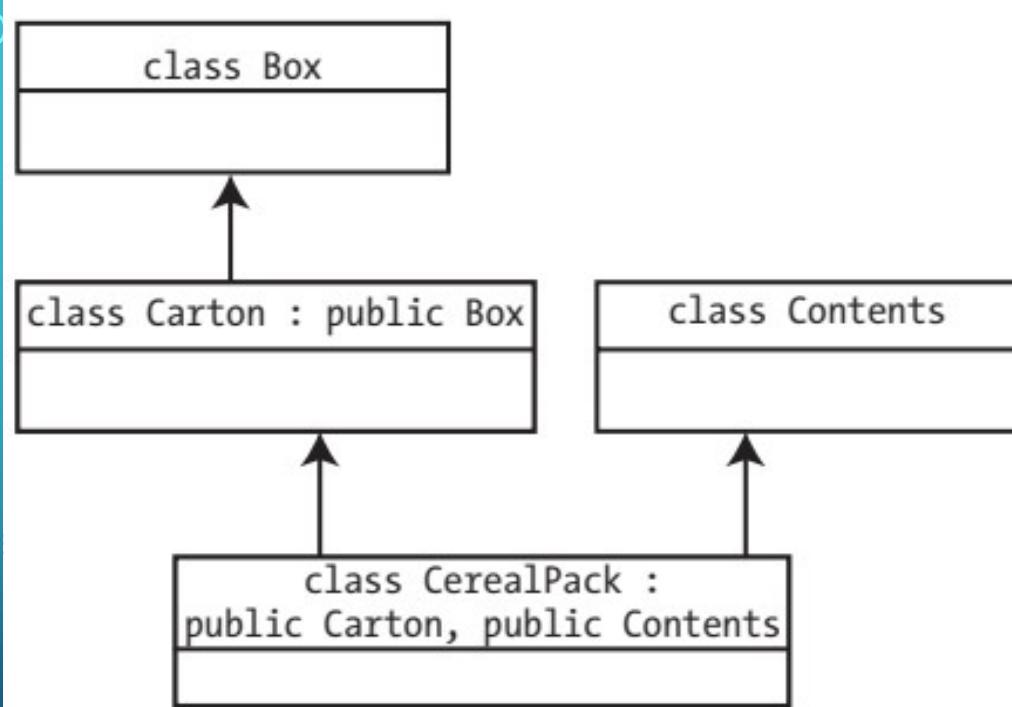
Ejercicio: Implemente la siguiente relación de clases. Los campos xSpeed e ySpeed simplemente mueven el punto a nuevas coordenadas.



POLIMORFISMO

Cuando tocamos el tema de herencia, vio cómo un objeto de un tipo de clase derivado contiene un subobjeto del tipo de clase base. En otras palabras, puede considerar cada objeto de clase derivado como un objeto de clase base. Debido a esto, siempre puede usar un puntero a una clase base para almacenar la dirección de un objeto de clase derivado; de hecho, puede usar un puntero a cualquier clase base directa o indirecta para almacenar la dirección de un objeto de clase derivado. La siguiente figura muestra cómo la clase Carton se deriva de la clase base Box por herencia simple, y la clase CerealPack se deriva por herencias múltiples de las clases base Carton y Contents. Ilustra cómo se pueden usar punteros a clases base para almacenar direcciones de objetos de clase derivados.

POLIMORFISMO



```
CerealPack breakfast;
// Puede almacenar la dirección
// de objeto breakfast
// en un puntero de tipo clase base

Carton* pCarton {&breakfast};
Box* pBox {&breakfast};
Contents* pContents {&breakfast};

// Puede almacenar la dirección
// de un objeto tipo Carton
// en un puntero de tipo clase base (Box)

Carton carton;
pBox = &carton;
```

POLIMORFISMO

Lo opuesto no es verdad. Por ejemplo, no puede usar un puntero de tipo Cartón* para almacenar la dirección de un objeto de tipo Box. Esto es lógico porque un tipo de puntero incorpora el tipo de objeto al que puede apuntar. Un objeto de clase derivado es una especialización de su base (es un objeto de clase base), por lo que es razonable usar un puntero a la base para almacenar su dirección. Sin embargo, un objeto de clase base definitivamente no es un objeto de clase derivado, por lo que un puntero a un tipo de clase derivado no puede señalarlo. Una clase derivada siempre contiene un subobjeto completo de cada una de sus bases, pero cada clase base representa solo una parte de un objeto de clase derivada.

POLIMORFISMO

Veremos un ejemplo específico. Suponga que deriva dos clases de la clase Box para representar diferentes tipos de contenedores, Carton y ToughPack. Supongamos además que el volumen de cada uno de estos tipos derivados se calcula de manera diferente. Para una caja de cartón, puede reducir el volumen ligeramente para tener en cuenta el grosor del material. Para un objeto ToughPack, es posible que deba reducir el volumen utilizable en una cantidad considerable para permitir el embalaje protector. La definición de las clases Carton y ToughPack podrían tener la siguiente forma:

```
class Carton : public Box
{
// Detalles de la clase

public:
    double volume() const;
};
```

```
class ToughPack : public Box
{
// Detalles de la clase..

public:
    double volume() const;
};
```

POLIMORFISMO

Dadas estas definiciones de clase (las definiciones de métodos siguen más adelante), puede declarar e inicializar un puntero de la siguiente manera:

```
Carton carton {10.0, 10.0, 5.0};  
Box* pBox {&carton};
```

El puntero pBox, de tipo puntero a Box, se ha inicializado con la dirección de carton. Esto es posible porque Carton se deriva de Box y, por lo tanto, contiene un subobjeto de tipo Box. Podría usar el mismo puntero para almacenar la dirección de un objeto ToughPack porque la clase ToughPack también se deriva de Box:

```
ToughPack hardcase {12.0, 8.0, 4.0};  
pBox = &hardcase;
```

POLIMORFISMO

El puntero pBox puede contener la dirección de cualquier objeto de cualquier clase que tenga Box como base. El tipo del puntero, Box*, se denomina tipo estático. Debido a que pBox es un puntero a una clase base, también tiene un tipo dinámico, que varía según el tipo de objeto al que apunta. Cuando pBox apunta a un objeto Carton, su tipo dinámico es un puntero a Carton. Cuando pBox apunta a un objeto ToughPack, su tipo dinámico es un puntero a ToughPack. Cuando pBox apunta a un objeto de tipo Box, su tipo dinámico es el mismo que su tipo estático. De ahí surge la magia del polimorfismo. Bajo las condiciones que explicaremos en breve, puede usar el puntero pBox para llamar a una función que está definida tanto en la clase base como en cada clase derivada y tener la función que realmente se llama seleccionada en tiempo de ejecución sobre la base del tipo dinámico de pBox. Considere estas sentencias:

```
double vol {};
vol = pBox->volume(); // Almacena el volumen del objeto al que apunta
```

POLIMORFISMO

Si pBox contiene la dirección de un objeto Carton, entonces esta sentencia llama a volume() para el objeto Carton. Si apunta a un objeto ToughPack, esta declaración llama a volume() para ToughPack. Esto funciona para cualquier clase derivada de Box, siempre que, por supuesto, se cumplan las condiciones antes mencionadas. Si lo son, la expresión pBox->volume() puede resultar en un comportamiento diferente dependiendo de a qué apunta pBox. Quizás lo más importante es que el comportamiento apropiado para el objeto al que apunta pBox se selecciona automáticamente en tiempo de ejecución. Con frecuencia surgen situaciones en las que el tipo específico de un objeto no se puede determinar de antemano, ni en tiempo de diseño ni en tiempo de compilación. Situaciones, en otras palabras, en las que el tipo se puede determinar solo en tiempo de ejecución.

POLIMORFISMO

Llamar a funciones heredadas

Antes de llegar a los detalles del polimorfismo, debemos explicar un poco más el comportamiento de las funciones miembro heredadas. Para ayudar con esto, revisaremos la clase Box para incluir una función que calcule el volumen de un objeto Box y otra función que muestre el volumen resultante. La nueva versión de la definición de clase en Box.h y Box.cpp será la siguiente:

POLIMORFISMO

Llamar a funciones heredadas

```
#ifndef BOX_H
#define BOX_H

#include <iostream>

class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    Box() = default;
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {}
        // Método que muestra el volumen de un objeto
    void showVolume() const
    { std::cout << "Volumen utilizable de Box:" << volume() << std::endl; }

        // Método que calcula el volumen de un objeto.
    double volume() const { return length * width * height; }
};

#endif
```

POLIMORFISMO

Llamar a funciones heredadas

Podemos mostrar el volumen utilizable de un objeto Box llamando a la función `showVolume()` para el objeto. Las campos se especifican como protegidos, por lo que las métodos de cualquier clase derivada pueden acceder a ellas. También definiremos la clase `ToughPack` con `Box` como base. Un objeto `ToughPack` incorpora material de embalaje para proteger su contenido, por lo que su capacidad es solo el 85 por ciento de un objeto `Box` básico. Por lo tanto, se necesita un método de `volumen()` diferente en la clase derivada para dar cuenta de esto:

```
class ToughPack : public Box
{
public:
    // Constructor
    ToughPack(double lv, double wv, double hv) : Box {lv, wv, hv} {}

    // Método que calcula el volumen de un ToughPack permitiendo 15% para empaques
    double volume() const { return 0.85 * length * width * height; }
};
```

POLIMORFISMO

Llamar a funciones heredadas

Possiblemente, podría tener miembros adicionales en esta clase derivada, pero por el momento, lo mantendremos simple, concentrándonos en cómo funcionan las funciones heredadas. El constructor de la clase derivada simplemente llama el constructor de la clase base en su lista de inicializadores de miembros para establecer los valores de las variables miembro. No necesita ninguna declaración en el cuerpo del constructor de la clase derivada. También tiene una nueva versión de la función de volume () para reemplazar la versión de la clase base. La idea aquí es que puede obtener la función heredada showVolume() para llamar a la versión de clase derivada de volumen() cuando la llama para un objeto de la clase ToughPack. Veamos si funciona:

POLIMORFISMO

Llamar a funciones heredadas

```
include "Box.h"
#include "ToughPack.h"

int main()
{
    Box box {20.0, 30.0, 40.0};

    ToughPack hardcase {20.0, 30.0, 40.0};

    box.showVolume();
    hardcase.showVolume();
}
```

Salida obtenida:

```
Volumen utilizable de Box: 24000
Volumen utilizable de Box: 24000
```

POLIMORFISMO

Llamar a funciones heredadas

Se supone que el objeto de la clase derivada tiene una capacidad menor que el objeto de la clase base, por lo que obviamente el programa no funciona según lo previsto. Intentemos establecer qué es lo que está fallando. La segunda llamada a `showVolume()` en `main()` es para un objeto de la clase derivada, `ToughPack`, pero evidentemente esto no se está teniendo en cuenta. El volumen de un objeto `ToughPack` debe ser el 85 por ciento del de un objeto `Box` básico con las mismas dimensiones. El problema es que cuando la función `showVolume()` llama a la función `volume()`, el compilador la establece como la versión de `volumen()` definida en la clase base.

POLIMORFISMO

Llamar a funciones heredadas

No importa cómo llame a `showVolume()`, nunca llamará a la versión `ToughPack` de la función `volume()`. Cuando las llamadas a métodos se fijan de esta manera antes de que se ejecute el programa, se denomina *resolución estática* de la llamada a función o *enlace estático*. El término *enlace temprano (early binding)* también se usa comúnmente. En este ejemplo, una función particular de `volumen()` está vinculada a la llamada de la función `showVolume()` cuando el programa se compila y se vincula. Cada vez que se llama a `showVolume()`, utiliza la función `volume()` de la clase base que está vinculada a él.

POLIMORFISMO

Llamar a funciones heredadas

El mismo tipo de resolución ocurriría en la clase derivada ToughPack.

Si agrega una función showVolume() que llama a volume() a la clase ToughPack, la llamada a volume() se resuelve estáticamente en la función de clase derivada.

¿Qué sucede si llama directamente a la función de volume() para el objeto ToughPack? Como otro experimento, agreguemos sentencias en main() para llamar a la función volume() de un objeto ToughPack directamente y también a través de un puntero a la clase base:

```
std::cout << "Volumen de hardcase: " << hardcase.volume() << std::endl;  
Box *pBox {&hardcase};  
std::cout << "Volumen de hardcase: " << pBox->volume() << std::endl;
```

POLIMORFISMO

Llamar a funciones heredadas

Esto es bastante informativo. Puede ver que una llamada a `volume()` para el objeto de clase derivado, `hardcase`, llama al método `volume()` de clase derivada, que es lo que desea. Sin embargo, la llamada a través del puntero de la clase base `pBox` se resuelve en la versión de la clase base de `volume()`, aunque `pBox` contiene la dirección de `hardcase`. En otras palabras, ambas llamadas se resuelven estáticamente. El compilador implementa estas llamadas de la siguiente manera:

```
std::cout << "Volumen de hardcase: " << hardcase.ToughPack::volume() << std::endl;
Box *pBox {&hardcase};
std::cout << "Volumen de hardcase a través pBox: " << pBox->Box::volume() << std::endl;
```

POLIMORFISMO

Llamar a funciones heredadas

Una llamada estática a función a través de un puntero está determinada únicamente por el tipo de puntero y no por el objeto al que apunta. En otras palabras, está determinado por el tipo estático en lugar del tipo dinámico. El puntero pBox tiene un puntero de tipo estático a Box, por lo que cualquier llamada estática que use pBox solo puede llamar a una función miembro de Box.

Cualquier llamada a una función a través de un puntero de clase base que se resuelve estáticamente, llama a una función de clase base.

POLIMORFISMO

Llamar a funciones heredadas

Lo que queremos es que el método `volume()` que se va a llamar en cualquier instancia dada se resuelva cuando se ejecuta el programa. Por lo tanto, si se llama a `showVolume()` para un objeto de clase derivada, queremos que se llame al método `volumen()` de la clase derivada, no a la versión de clase base. Cuando se llama al método `volume()` a través de un puntero de clase base, queremos que se llame al método `volumen()` que sea apropiado para el objeto al que se apunta. Este tipo de operación se conoce como *enlace dinámico* o *enlace tardío*. Para que esto funcione, tenemos que decirle al compilador que el método `volume()` en `Box` y cualquier anulación en las clases derivadas de `Box` son especiales, y las llamadas a ellas deben resolverse dinámicamente. Podemos obtener este efecto especificando que `volume()` en la clase base es una función virtual, lo que resultará en una llamada de función virtual para `volume()`.

POLIMORFISMO

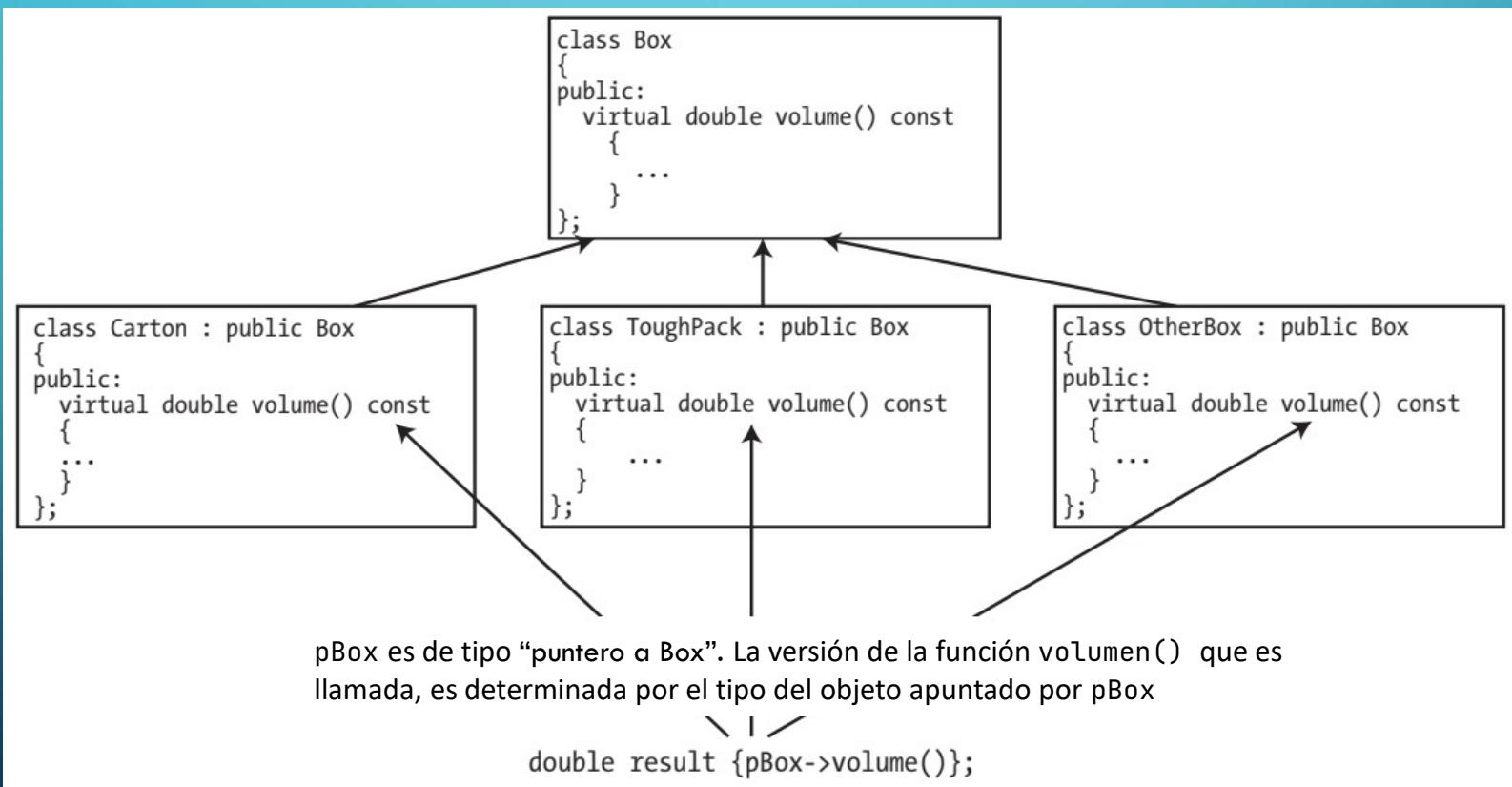
Funciones Virtuales

Cuando especifica una función como **virtual** en una clase base, le indica al compilador que desea un **enlace dinámico** para las llamadas a métodos en cualquier clase que se derive de esta clase base. Una función virtual se declara en una clase base usando la palabra clave **virtual**, como se muestra en la siguiente figura. Describir una clase como polimórfica significa que es una clase derivada que contiene al menos una función virtual.

POLIMORFISMO

Funciones Virtuales

Como llamar a funciones virtuales



POLIMORFISMO

Funciones Virtuales

Una función que especifique como `virtual` en una clase base será `virtual` en todas las clases que se derivan directa o indirectamente de la base. Este es el caso ya sea que especifique o no la función como `virtual` en una clase derivada. *Para obtener un comportamiento polimórfico, cada clase derivada puede implementar su propia versión de la función virtual (aunque no está obligado a hacerlo, lo veremos más adelante).* Puede realizar llamadas a funciones virtuales utilizando una variable cuyo tipo es un puntero o una referencia a un objeto de clase base. La figura anterior ilustra cómo se resuelve dinámicamente una llamada a una función virtual a través de un puntero. El puntero al tipo de clase base se utiliza para almacenar la dirección de un objeto con un tipo correspondiente a una de las clases derivadas.

POLIMORFISMO

Funciones Virtuales

Podría apuntar a un objeto de cualquiera de las tres clases derivadas mostradas o, por supuesto, a un objeto de clase base. El tipo de objeto al que apunta el puntero cuando se ejecuta la llamada determina qué función de `volume ()` se llama. Tenga en cuenta que una llamada a una función virtual mediante un objeto siempre se resuelve de forma estática. **Solo obtiene resolución dinámica de llamadas a funciones virtuales a través de un puntero o una referencia.** Almacenar un objeto de un tipo de clase derivada en una variable de un tipo base dará como resultado que el objeto de la clase derivada se divida, por lo que no tiene características de clase derivada. Dicho esto, demos un giro a las funciones virtuales. Para que el ejemplo anterior funcione como debería, se requiere un cambio muy pequeño en la clase `Box`. Solo necesitamos agregar la palabra clave `virtual` a la definición de la función `volumen ()`:

POLIMORFISMO

Funciones Virtuales

```
class Box
{
    // Resto de la clase..

public:
    // Función para calcular el volumen del objeto Box
    virtual double volume() const { return length * width * height; }
};
```

CUIDADO: si una definición de método está fuera de la definición de clase, no debe agregar la palabra clave `virtual` a la definición del método; sería un error hacerlo. Solo puede agregar `virtual` a declaraciones o definiciones dentro de una definición de clase.

Para hacerlo más interesante, implementemos el método `volume()` en una nueva clase llamada `Carton` de forma un poco diferente. Aquí está la definición de clase:

POLIMORFISMO

Funciones Virtuales

```
// Carton.h
#ifndef CARTON_H
#define CARTON_H

#include <string>
#include <string_view>
#include "Box.h"

class Carton : public Box
{
private:
    std::string material;

public:
    // Constructor llama explicitamente al constructor de la clase base
    Carton(double lv, double wv, double hv, std::string_view str="cardboard")
        : Box{lv,wv,hv}, material{str}
    {}

    // Método para calcular el volumen del objeto Carton
    double volume() const
    {
        const double vol { (length - 0.5)*(width - 0.5)*(height - 0.5)};
        return vol > 0.0? vol : 0.0;
    }
};

#endif
```

POLIMORFISMO

Funciones Virtuales

La función `volume()` para un objeto Cartón supone que el grosor del material es 0.25, por lo que se resta 0.5 de cada dimensión para tener en cuenta los lados de la caja. Si se ha creado un objeto Cartón con alguna de sus dimensiones inferior a 0.5 por algún motivo, esto dará como resultado un valor negativo para el volumen, por lo que, en tal caso, el volumen del cartón se establecerá en cero. También usaremos la clase ToughPack. Aquí está el código para el archivo fuente que contiene `main()`:

POLIMORFISMO

Funciones Virtuales

```
// Uso de funciones virtuales
#include <iostream>
#include "Box.h"
#include "ToughPack.h"
#include "Carton.h"

int main()
{
    Box box {20.0, 30.0, 40.0};
    ToughPack hardcase {20.0, 30.0, 40.0}; // Box derivado - mismo tamaño
    Carton carton {20.0, 30.0, 40.0, "plastic"}; // Otro box derivado

    box.showVolume(); // Volumen de box
    hardcase.showVolume(); // Volumen de ToughPack
    carton.showVolume(); // Volumen de Carton

    // Ahora usamos un puntero de tipo clase Base
    Box* pBox {&box}; // Apunta un tipo Box
    std::cout << "\nVolumen de a través de box Box: " << pBox->volume() << std::endl;
    pBox->showVolume();

    pBox = &hardcase; // Apunta a tipo ToughPack
    std::cout << "Volumen de hardcase a través de pBox: " << pBox->volume() << std::endl;
    pBox->showVolume();

    pBox = &carton; // Apunta a tipo Carton
    std::cout << "Volumen de carton a través de pBox: " << pBox->volume() << std::endl;
    pBox->showVolume();
}
```

EJEMPLO 02:

POLIMORFISMO

Funciones Virtuales

```
Volumen utilizable de Box: 24000  
Volumen utilizable de Box: 20400  
Volumen utilizable de Box: 22722.4
```

Salida de EJEMPLO 02 :

```
Volumen de box a través de pBox: 24000  
Volumen utilizable de Box: 24000  
Volumen de hardcase a través de pBox: 20400  
Volumen utilizable de Box: 20400  
Volumen de carton a través de pBox: 22722.4  
Volumen utilizable de Box: 22722.4
```

Tenga en cuenta que no hemos agregado la palabra clave virtual a las funciones volume () de la clase Carton o ToughPack. La palabra clave virtual aplicada a la función volumen () en la clase base es suficiente para determinar que todas las definiciones de la función en las clases derivadas también serán virtuales. Opcionalmente, también puede usar la palabra clave virtual para sus funciones de clase derivadas. Si lo hace o no es una cuestión de preferencia personal. Volveremos a esta elección más adelante. El programa ahora claramente está haciendo lo que queríamos.

POLIMORFISMO

Funciones Virtuales

La llamada a `showVolume()` para el objeto `box` llama a la versión de clase base de `volume()` porque `box` es de tipo `Box`. La próxima llamada a `showVolume()` es para el estuche rígido del objeto `ToughPack`. Llama a la función `showVolume()` heredada de la clase `Box`, pero la llamada a `volume()` en `showVolume()` se resuelve en la versión definida en la clase `ToughPack` porque `volume()` es una función virtual. Por lo tanto, obtiene el volumen calculado adecuadamente para un objeto `ToughPack`. La tercera llamada de `showVolume()` para el objeto `carton` llama a la versión de la clase `Carton` de `volume()`, por lo que también obtendrá el resultado correcto.

POLIMORFISMO

Funciones Virtuales

A continuación, utiliza el puntero pBox para llamar a la función `volume()` directamente y también indirectamente a través de la función no virtual `showVolume()`. El puntero contiene primero la dirección del objeto Box y luego las direcciones de los dos objetos de clase derivados a su vez. El resultado de cada objeto muestra que la versión adecuada de la función `volume()` se selecciona automáticamente en cada caso, por lo que tiene una demostración clara del polimorfismo en acción.

POLIMORFISMO

Funciones Virtuales

Para que una función se comporte “virtualmente”, su definición en una clase derivada debe tener la misma firma que tiene en la clase base. Si la función de clase base es `const`, por ejemplo, entonces la función de clase derivada también debe ser `const`. Generalmente, el tipo de retorno de una función virtual en una clase derivada también debe ser el mismo que el de la clase base, **pero hay una excepción cuando el tipo de retorno en la clase base es un puntero o una referencia a un tipo de clase**. En este caso, la versión de clase derivada de una función virtual puede devolver un puntero o una referencia a un tipo más especializado que el de la base. No profundizaremos en esto, pero en caso de que lo encuentre en otro lugar, el término técnico utilizado en relación con estos tipos de devolución es covarianza.

POLIMORFISMO

Funciones Virtuales

Si el nombre de la función y la lista de parámetros de una función en una clase derivada son los mismos que los de una función virtual declarada en la clase base, el tipo de valor devuelto debe ser coherente con las reglas de una función virtual. Si no es así, la función de clase derivada no se compilará. Otra restricción es que una función virtual no puede ser una función de plantilla (*template*).

Una función en una clase derivada que redefine una función virtual de la clase base anula (*override*) esta función. Una función con el mismo nombre que una función virtual en una clase base solo anula esa función si el resto de sus firmas también coinciden exactamente; si no lo hacen, la función en la clase derivada es una nueva función que oculta la de la clase base.

POLIMORFISMO

Funciones virtuales

Requisitos para la operación de funciones virtuales

Esto último implica nombres de funciones miembro duplicadas. Por supuesto, esto implica que si intenta usar diferentes parámetros para una función virtual en una clase derivada o usa diferentes especificadores constantes, entonces el mecanismo de la función virtual no funcionará. La función en la clase derivada luego define una función nueva y diferente, y esta nueva función, por lo tanto, operará con un enlace estático que se establece y corrige en el momento de la compilación.

Puede probar esto eliminando la palabra clave `const` de la definición de `volume()` en la clase `Cartón` y ejecutando el ejemplo nuevamente. La firma de la función `volume()` en `Carton` ya no coincide con la función virtual en `Box`, por lo que la función `volume()` de la clase derivada no es virtual. En consecuencia, la resolución es estática, de modo que la función llamada para los objetos `Carton` a través de un puntero base, o incluso indirectamente a través de la función `showVolume()`, es la versión de la clase base.

POLIMORFISMO

Funciones virtuales

Requisitos para la operación de funciones virtuales

Las métodos estáticas no pueden ser virtuales. Esto nos da otra razón más para llamar siempre a las funciones miembro estáticas prefijándolas con el nombre de la clase en lugar del de un objeto. es decir, utilice siempre MiClase::miFuncionEstatica() en lugar de miObjeto.miFuncionEstatica(). esto deja muy claro que no se debe esperar polimorfismo.

POLIMORFISMO

Funciones Virtuales - override

Es fácil cometer un error en la especificación de una función virtual en una clase derivada. Si define `Volume()` (observe la V mayúscula) en una clase derivada de `Box`, no será virtual porque la función virtual en la clase base es `volume()`. Esto significa que las llamadas a `Volume()` se resolverán de forma estática y la función de volumen `virtual()` en la clase se heredará de la clase base. El código aún puede compilarse y ejecutarse, pero no correctamente. De manera similar, si define una función de `volume()` en una clase derivada pero olvida especificar `const`, esta función se sobre cargará en lugar de anular la función de la clase base. Este tipo de errores pueden ser difíciles de detectar. Puede protegerse contra tales errores usando el especificador de anulación (`override`) para cada declaración de función virtual en una clase derivada, como esta:

POLIMORFISMO

Funciones Virtuales - override

```
class Carton : public Box
{
// Detalles de la clase como en el ejemplo..

public:
    double volume() const override
    {
        // Cuerpo del método volume.
    }
};
```

La especificación de anulación (**override**), como la **virtual**, solo aparece dentro de la definición de clase. No debe aplicarse a una definición externa de un método. La especificación de anulación hace que el compilador verifique que la clase base declara un miembro de clase que es **virtual** y tiene la misma firma. Si no es así, el compilador marca la definición aquí como un error (¡pruébelo!).

POLIMORFISMO

Funciones Virtuales - final

En ocasiones, es posible que desee evitar que una función miembro se invalide en una clase derivada. Esto podría deberse a que desea limitar cómo una clase derivada puede modificar el comportamiento de la interfaz de clase, por ejemplo. Puede hacer esto especificando que una función es final. Puede evitar que el método `volume()` en la clase `Carton` sea anulada por definiciones en clases derivadas de `Carton` especificándola de esta manera:

```
class Carton : public Box
{
// Detalles de la clase como en el ejemplo
public:
    double volume() const override final
    {
        // Cuerpo de la función volume.
    }
};
```

POLIMORFISMO

Funciones Virtuales - final

Los intentos de anular `volume()` en clases que tienen `Cartón` como base darán como resultado un error de compilación. Esto garantiza que solo se pueda usar la versión `Carton` para objetos de clase derivados. El orden en el que coloca las palabras clave `override` y `final` no importa, por lo que tanto `override final` como `final override` son correctos, pero ambas deben ir después de `const` o cualquier otra parte de la firma de la función.

También podemos declarar la clase entera como `final`:

```
class Carton final : public Box
{
    // Detalles de la clase
public:
    double volume() const override
    {
        // Cuerpo del método volume
    }
};
```

POLIMORFISMO

Funciones Virtuales - final

Ahora el compilador no permitirá que Carton se use como clase base. No es posible derivar más de la clase Carton. Tenga en cuenta que esta vez es perfectamente sensato usar final en una clase que no tiene ninguna clase base propia. Sin embargo, lo que no tiene sentido es introducir nuevas funciones virtuales en una clase final, es decir, funciones virtuales que no invaliden una función de clase base.

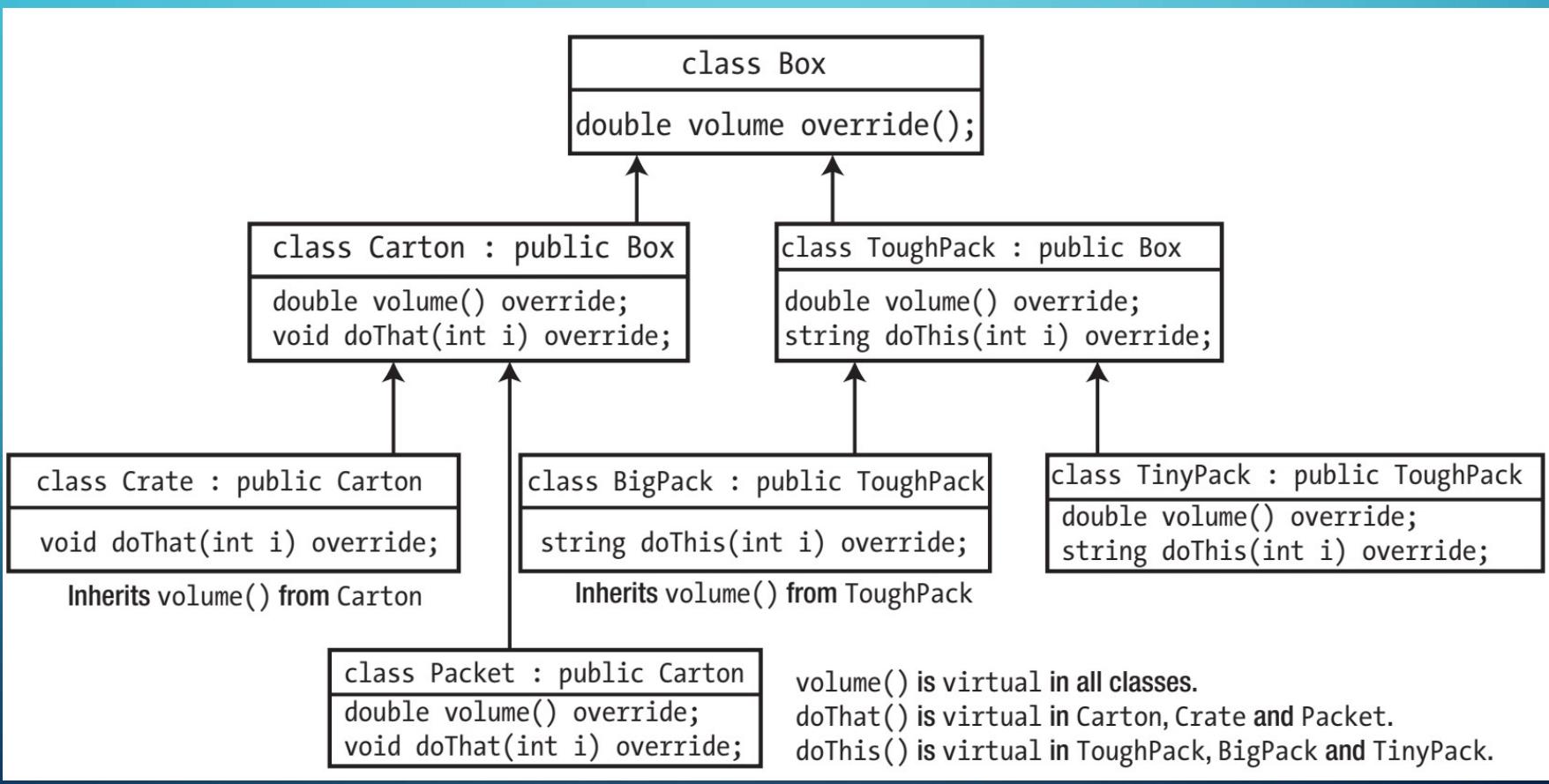
POLIMORFISMO

Funciones Virtuales - Funciones virtuales y jerarquías de clases

Si desea que su función se trate como virtual cuando se llama mediante un puntero de clase base, debe declararla como virtual en la clase base. Puede tener tantas funciones virtuales como desee en una clase base, pero no todas las funciones virtuales deben declararse dentro de la clase base más básica en una jerarquía. Esto se ilustra en la siguiente figura:

POLIMORFISMO

Funciones Virtuales - Funciones virtuales y jerarquías de clases



POLIMORFISMO

Funciones Virtuales - Funciones virtuales y jerarquías de clases

Cuando especifica una función como virtual en una clase, la función es virtual en todas las clases derivadas directa o indirectamente de esa clase. Todas las clases derivadas de la clase Box de la figura anterior heredan la naturaleza virtual de la función `volume()`, incluso si no repiten la palabra clave `virtual`.

Puede llamar a `volume()` para objetos de cualquiera de estos tipos de clase a través de un puntero de tipo `Box*` porque el puntero puede contener la dirección de un objeto de cualquier clase en la jerarquía. La clase Crate no define el `volume()`, por lo que la versión heredada de Carton se llamaría para los objetos Crate. Se hereda como una función virtual y, por lo tanto, se puede llamar polimórficamente.

POLIMORFISMO

Funciones Virtuales - Funciones virtuales y jerarquías de clases

Un puntero pCarton, de tipo Carton*, también podría usarse para llamar a volume(), pero solo para objetos de la clase Carton y las dos clases que tienen Carton como base: Crate y Packet. La clase Carton y las clases derivadas de ella también contienen la función virtual doThat(). Esta función también se puede llamar polimórficamente usando un puntero de tipo Carton*. Por supuesto, no puede llamar a doThat() para estas clases usando un puntero de tipo Box* porque la clase Box no define la función doThat(). De manera similar, la función virtual doThis() podría llamarse para objetos de tipo ToughPack, BigPack y TinyPack utilizando un puntero de tipo ToughPack*. Por supuesto, el mismo puntero también podría usarse para llamar a la función volume() para objetos de estos tipos de clase.

POLIMORFISMO

Especificadores de acceso y funciones virtuales

La especificación de acceso de una función virtual en una clase derivada puede ser diferente de la especificación en la clase base. Cuando llama a la función virtual a través de un puntero de clase base, la especificación de acceso en la clase base determina si la función es accesible, independientemente del tipo de objeto al que apunta. Si la función virtual es pública en la clase base, se puede llamar para cualquier clase derivada a través de un puntero (o una referencia) a la clase base, independientemente de la especificación de acceso en la clase derivada. Podemos demostrar esto modificando el ejemplo anterior. Modifique la definición de la clase ToughPack para proteger la función volume() y agregue la palabra clave override a su declaración para asegurarse de que realmente anula una función virtual de la clase base:

POLIMORFISMO

Especificadores de acceso y funciones virtuales

```
class ToughPack : public Box
{
public:
    // Constructor
    ToughPack(double lv, double wv, double hv) : Box {lv, wv, hv} {}

protected:
    // Método para calcular el volumen de un ToughPack dejando 15% para empaques
    double volume() const override { return 0.85 * length * width * height; }
};
```

POLIMORFISMO

```
// Especificadores de acceso y funciones virtuales // EJEMPLO 03
#include <iostream>
#include "Box.h"
#include "ToughPack.h"
#include "Carton.h"

int main()
{
    Box box {20.0, 30.0, 40.0};
    ToughPack hardcase {20.0, 30.0, 40.0}; // Clase derivada de box - mismo tamaño
    Carton carton {20.0, 30.0, 40.0, "plastic"}; // Otra clase derivada de box

    box.showVolume(); // Volumen de Box
    hardcase.showVolume(); // Volumen de ToughPack
    carton.showVolume(); // Volumen de Carton

    // Descomentar la siguiente línea para producir un error
    // std::cout << "Volumen de hardcase: " << hardcase.volume() << std::endl;

    // Ahora utilizamos punteros.
    Box* pBox {&box}; // Apunta a tipo Box
    std::cout << "\nVolumen de box a través de pBox: " << pBox->volume() << std::endl;
    pBox->showVolume();

    pBox = &hardcase; // Apunta a tipo ToughPack
    std::cout << "Volumen de hardcase a través de pBox: " << pBox->volume() << std::endl;
    pBox->showVolume();

    pBox = &carton; // Apunta a tipo Carton
    std::cout << "Volumen de carton a través de pBox: " << pBox->volume() << std::endl;
    pBox->showVolume();
}
```

EJEMPLO 03:

POLIMORFISMO

Especificadores de acceso y funciones virtuales

No debería sorprender que este código produzca el mismo resultado que el último ejemplo. Aunque `volume()` se declara como protegido en la clase `ToughPack`, aún puede llamarlo para el objeto `hardcase` a través de la función `showVolume()` que se hereda de la clase `Box`. También puede llamarlo directamente a través de un puntero a la clase base, `pBox`. Sin embargo, si quita el comentario de la línea que llama a la función de `volumen()` directamente usando el objeto `hardcase`, el código no se compilará.

Lo que importa aquí es si la llamada se resuelve de forma dinámica o estática. Cuando usa un objeto de clase, el compilador determina estáticamente la llamada. Llamar a `volume()` para un objeto `ToughPack` llama a la función definida en esa clase. Debido a que el método `volume()` está protegida en `ToughPack`, la llamada al objeto `hardcase` no se compilará.

POLIMORFISMO

Especificadores de acceso y funciones virtuales

Todas las demás llamadas se resuelven cuando se ejecuta el programa; son llamadas polimórficas. En este caso, la especificación de acceso para una función virtual en la clase base se hereda en todas las clases derivadas. Esto es independiente de la especificación explícita en la clase derivada; la especificación explícita solo afecta a las llamadas que se resuelven estáticamente. Por lo tanto, los especificadores de acceso determinan si se puede llamar a una función según el tipo estático de un objeto. La consecuencia es que cambiar el especificador de acceso de una anulación de función a uno más restringido que el de la función de clase base es algo inútil. Esta restricción de acceso se puede eludir fácilmente mediante el uso de un puntero a la clase base. Esto se muestra mediante la función `showVolume()` de `ToughPack` en el ejemplo 03.

POLIMORFISMO

Uso de referencias para llamar a funciones virtuales

Puede llamar a una función virtual a través de una referencia. Los parámetros por referencia son herramientas particularmente poderosas para aplicar polimorfismo, particularmente cuando se llama a funciones que usan pass-by-reference. Puede pasar un objeto de clase base o cualquier objeto de clase derivado a una función con un parámetro que sea una referencia a la clase base. Puede usar el parámetro de referencia dentro del cuerpo de la función para llamar a una función virtual en la clase base y obtener un comportamiento polimórfico. Cuando se ejecuta la función, la función virtual para el objeto que se pasó como argumento se selecciona automáticamente en tiempo de ejecución. Podemos mostrar esto en acción modificando el Ejemplo 02 para llamar a una función que tiene un parámetro de tipo referencia a Box:

POLIMORFISMO

Uso de referencias para llamar a funciones virtuales

```
// Utilizando un parametro por referencia para llamar a una función virtual
#include <iostream>
#include "Box.h"
#include "ToughPack.h"
#include "Carton.h"

// Función global para imprimir el volumen de una caja
void showVolume(const Box& rBox)
{
    std::cout << "Volumen utilizable de Box:" << rBox.volume() << std::endl;
}

int main()
{
    Box box {20.0, 30.0, 40.0}; // Un box base
    ToughPack hardcase {20.0, 30.0, 40.0}; // Un box derivado - mismo tamaño
    Carton carton {20.0, 30.0, 40.0, "plastic"}; // Otro box derivado

    showVolume(box); // Mostrar volumen de un box de clase base
    showVolume(hardcase); // Mostrar volumen de un box de clase derivada
    showVolume(carton); // Mostrar volumen de un box de clase derivada
}
```

EJEMPLO 05:

POLIMORFISMO

Uso de referencias para llamar a funciones virtuales

Salida EJEMPLO 05

```
Volumen utilizable de Box: 24000  
Volumen utilizable de Box: 20400  
Volumen utilizable de Box: 22722.4
```

Las definiciones de clase son las mismas que en EJEMPLO 02. Hay una nueva función global que llama a `volume()` usando su parámetro de referencia para llamar al miembro `volume()` de un objeto. `main()` define los mismos objetos que en EJEMPLO 02 pero llama a la función global `showVolume()` con cada uno de los objetos para generar sus volúmenes. Como puede ver en el resultado, se usa la función de `volume()` correcta en cada caso, lo que confirma que el polimorfismo funciona a través de un parámetro de referencia. Cada vez que se llama a la función `showVolume()`, el parámetro de referencia se inicializa con el objeto que se pasa como argumento. Debido a que el parámetro es una referencia a una clase base, el compilador organiza el enlace dinámico a la función virtual `volume()`.

POLIMORFISMO

Colecciones polimórficas

El polimorfismo se vuelve particularmente interesante cuando se trabaja con las llamadas colecciones polimórficas o heterogéneas de objetos, ambos nombres sofisticados para colecciones de punteros de clase base que contienen objetos con diferentes tipos dinámicos. Los ejemplos de colecciones incluyen arrays simples de estilo C, pero también las plantillas `std::array<>` y `std::vector<>` más modernas y potentes de la Biblioteca estándar. Demostraremos este concepto usando las clases Box, Carton y ToughPack de EJEMPLO 03 y una función `main()` revisada:

POLIMORFISMO

Colecciones polimórficas

```
// Vectores polimórficos de punteros inteligentes
#include <iostream>
#include <memory>
#include <vector>
#include "Box.h"
#include "ToughPack.h"
#include "Carton.h"

int main()
{
    // Cuidado: este primer intento de una colección mixta es una mala idea (!corte de objetos!)
    std::vector<Box> boxes;
    boxes.push_back(Box{20.0, 30.0, 40.0});
    boxes.push_back(ToughPack{20.0, 30.0, 40.0});
    boxes.push_back(Carton{20.0, 30.0, 40.0, "plastic"});

    for (const auto& p : boxes)
        p.showVolume();

    std::cout << std::endl;

    // Ahora creamos un vector<> polimórfico propio:
    std::vector<std::unique_ptr<Box>> polymorphicBoxes;
    polymorphicBoxes.push_back(std::make_unique<Box>(20.0, 30.0, 40.0));
    polymorphicBoxes.push_back(std::make_unique<ToughPack>(20.0, 30.0, 40.0));
    polymorphicBoxes.push_back(std::make_unique<Carton>(20.0, 30.0, 40.0, "plastic"));

    for (const auto& p : polymorphicBoxes)
        p->showVolume();
}
```

EJEMPLO 06:

POLIMORFISMO

Colecciones polimórficas

Salida EJEMPLO 06:

```
Volumen utilizable de Box: 24000
```

```
Volumen utilizable de Box: 20400
```

```
Volumen utilizable de Box: 22722.4
```

La primera parte del programa muestra **cómo no crear** una colección polimórfica. Si asigna objetos de clases derivadas en un vector<> de objetos de clase base por valor, como siempre, se producirá la división de objetos (object slicing). Es decir, solo se conserva el subobjeto correspondiente a esa clase base. El vector en general no tiene espacio para almacenar el objeto completo. El tipo dinámico del objeto también se convierte en el de la clase base. **Si desea polimorfismo, sabe que siempre debe trabajar con punteros o referencias.**

POLIMORFISMO

Colecciones polimórficas

Para nuestro vector polimórfico adecuado en la segunda parte del programa, podríamos haber usado un `vector<>` de punteros `Box*` simples, es decir, un vector de tipo `std::vector<Box*>`, y almacenar punteros a objetos `Box`, `ToughPack` y `Carton` asignados dinámicamente allí. La desventaja de eso habría sido que hubiéramos tenido que recordar eliminar también estos objetos `Box` al final del programa.

Ya sabe que la Biblioteca estándar ofrece los llamados punteros inteligentes para ayudar con esto. Los punteros inteligentes nos permiten trabajar con seguridad con punteros sin tener que preocuparnos todo el tiempo por borrar los objetos.

POLIMORFISMO

Colecciones polimórficas

En el vector `polymorphicBoxes`, por lo tanto, almacenamos elementos de tipo `std::unique_ptr<Box>`, que son punteros inteligentes a objetos Box. Los elementos pueden almacenar direcciones para objetos de Box o cualquier clase derivada de Box, por lo que existe un paralelismo exacto con los punteros crudos que has visto hasta ahora. Afortunadamente, como muestra el resultado, el polimorfismo sigue vivo y bien con los punteros inteligentes. Cuando está creando objetos en el almacenamiento libre, los punteros inteligentes aún le brindan un comportamiento polimórfico y al mismo tiempo eliminan cualquier posible pérdida de memoria (memory leaks).

POLIMORFISMO

Colecciones polimórficas

Para obtener colecciones polimórficas de objetos *memory-safe*, puede almacenar punteros inteligentes estándar como `std::unique_ptr<>` y `shared_ptr<>` dentro de contenedores estándar como `std::vector<>` y `array<>`.

POLIMORFISMO

Destruir objetos a través de un puntero

El uso de punteros a una clase base cuando trabaja con objetos de clases derivados es muy común porque así es como puede aprovechar las funciones virtuales. Si usa punteros o punteros inteligentes para objetos creados en el almacenamiento libre, puede surgir un problema cuando se destruyen los objetos de clase derivados. Puede ver el problema si agrega destructores a las diversas clases de Box que muestran un mensaje. Comience desde los archivos de EJEMPLO 06 y agregue un destructor a la clase base de Box que solo muestra un mensaje cuando se llama:

```
~Box() { std::cout << "Destructor de Box llamado" << std::endl; }
```

POLIMORFISMO

Destruir objetos a través de un puntero

En la clase ToughPack también agregue un destructor de la siguiente forma:

```
~ToughPack() { std::cout << "Destructor de ToughPack llamado" << std::endl; }
```

En la clase Carton también agregue un destructor de la siguiente forma:

```
~Carton() { std::cout << "Destructor Carton llamado" << std::endl; }
```

```
Volumen utilizable de Box: 24000  
Volumen utilizable de Box: 24000  
Volumen utilizable de Box: 24000
```

Salida EJEMPLO 07:

```
Volumen utilizable de Box: 24000  
Volumen utilizable de Box: 20400  
Volumen utilizable de Box: 22722.4  
Destructor de Box llamado  
Destructor de Box llamado
```

POLIMORFISMO

Destruir objetos a través de un puntero

Claramente tenemos un fracaso en nuestras manos. Se llama al mismo destructor de clase base para los seis objetos, aunque cuatro de ellos son objetos de una clase derivada. Esto ocurre incluso para los objetos almacenados en el vector polimórfico. Naturalmente, la causa de este comportamiento es que la función destructora se resuelve estáticamente en lugar de dinámicamente, como cualquier otra función. Para garantizar que se llame al destructor correcto para una clase derivada, necesitamos un enlace dinámico para los destructores. Lo que necesitamos son funciones de destructor virtual.

Nota: el resultado que verá antes de la salida mostrada corresponde a los diversos elementos de Box que se empujan y cortan en el primer vector.

POLIMORFISMO

Destruir objetos a través de un puntero

CUIDADO: puede pensar que para objetos de clases como `ToughPack` o `Carton`, llamar al destructor incorrecto no es gran cosa porque sus destructores están básicamente vacíos. No es que los destructores de estas clases derivadas realicen ninguna tarea de limpieza crítica ni nada, entonces, ¿cuál es el daño si no se les llama? el quid de la cuestión es que el estándar C++ establece específicamente que aplicar `delete` en un puntero de clase base a un objeto de una clase derivada da como resultado un comportamiento indefinido, a menos que esa clase base tenga un destructor virtual. por lo tanto, si bien llamar al destructor incorrecto puede parecer inofensivo, incluso durante la ejecución del programa, en principio podría pasar cualquier cosa. Si tienes suerte, es benigno y no pasa nada malo. Pero también podría introducir fugas de memoria (tal vez solo se libera la memoria para el subobjeto de la clase base) o incluso bloquear su programa.

POLIMORFISMO

DESTRUCTORES VIRTUALES

Para asegurarse de que siempre se llame al destructor correcto para los objetos de clases derivadas que se asignan en el almacenamiento libre, necesita destructores de clases virtuales. Para implementar un destructor virtual en una clase derivada, simplemente agregue la palabra clave `virtual` a la declaración del destructor en la clase base. Esto le indica al compilador que las llamadas al destructor a través de un puntero o un parámetro de referencia deben tener un enlace dinámico, por lo que el destructor que se llama se seleccionará en tiempo de ejecución. Esto hace que el destructor en cada clase derivada de la clase base sea virtual, a pesar de que los destructores de la clase derivada tengan nombres diferentes; los destructores se tratan como un caso especial para este fin.

Para probar esto, modifique el EJEMPLO 07 haciendo los destructores virtuales. Para ello agregue la palabra clave `virtual` delante del nombre del constructor.

POLIMORFISMO

DESTRUCTORES VIRTUALES

Ejemplo para la clase Box:

```
virtual ~Box() { std::cout << "Destructor de Box llamado" << std::endl; }
```

Los destructores de todas las clases derivadas serán automáticamente virtuales como resultado de declarar un destructor de clase base virtual. Si vuelve a ejecutar el ejemplo, la salida confirmará que esto es así. Si no fuera por el mensaje de salida que agregamos con fines ilustrativos, el cuerpo de la función habría sido un bloque {} vacío. Sin embargo, en lugar de usar un bloque vacío de este tipo, le recomendamos que declare el destructor usando la palabra clave default. Esto hace que sea mucho más visible que se utiliza una implementación predeterminada. Para nuestra clase Box, escribiría lo siguiente:

```
virtual ~Box() = default;
```

POLIMORFISMO

DESTRUCTORES VIRTUALES

La palabra clave **default** se puede usar para todos los miembros que el compilador normalmente generaría para usted. Esto incluye destructores pero también, como vio anteriormente, constructores y operadores de asignación. Tenga en cuenta que los destructores generados por el compilador nunca son virtuales, a menos que los declare explícitamente como tales.

POLIMORFISMO

DESTRUCTORES VIRTUALES

Cuando se espera un uso polimórfico (o incluso es posible), su clase debe tener un destructor virtual para garantizar que sus objetos siempre se destruyan correctamente. Esto implica que tan pronto como una clase tenga al menos un método virtual, su destructor también debería volverse virtual. (La única vez que no tiene que seguir estas pautas es si el destructor no virtual es protected o private, pero estos son casos bastante excepcionales).

POLIMORFISMO

FUNCIONES VIRTUALES PURAS

Hay situaciones que requieren una clase base con varias clases derivadas de ella y una función virtual que se redefine en cada una de las clases derivadas, pero donde no hay una definición significativa para la función en la clase base. Por ejemplo, puede definir una clase base, Shape, de la que derive clases que definen formas específicas, como Círculo, Elipse, Rectángulo, Hexágono, etc. La clase Shape podría incluir una función virtual area() que llamaría para un objeto de clase derivado para calcular el área de una forma particular. Sin embargo, la clase Shape en sí misma no puede proporcionar una implementación significativa de la función area(), una que sirva, por ejemplo, tanto para círculos como para rectángulos. Este es un trabajo para una **función virtual pura**.

POLIMORFISMO

FUNCIONES VIRTUALES PURAS

El propósito de una función virtual pura es permitir que las versiones de clase derivadas de la función se llamen polimórficamente. Para declarar una función virtual pura en lugar de una función virtual "ordinaria" que tiene una definición, usa la misma sintaxis pero agrega = 0 a su declaración dentro de la clase. Si todo esto suena confuso en términos abstractos, puede ver cómo declarar una función virtual pura observando el ejemplo concreto de definición de la clase Shape a la que acabamos de aludir:

POLIMORFISMO

FUNCIONES VIRTUALES PURAS

```
// Clase base genérica para figuras
class Shape
{
protected:
    Point position; // Posicion de una figura
    Shape(const Point& shapePosition) : position {shapePosition} {}

public:
    virtual ~Shape() = default; // Siempre utilice destructores virtuales para las clases base
    virtual double area() const = 0; // Funcion virtual pura para calcular el area de una figura.
    virtual void scale(double factor) = 0; // Función virtual pura para escalar una figura

    // Función virtual regular para mover una figura
    virtual void move(const Point& newPosition) { position = newPosition; };
};
```

POLIMORFISMO

FUNCIONES VIRTUALES PURAS

La clase Shape contiene una variable miembro de tipo Point (que es otro tipo de clase) que almacena la posición de una forma. Es un miembro de la clase base porque cada figura debe tener una posición y el constructor de figuras la inicializa. Las funciones `area()` y `scale()` son virtuales porque están calificadas con la palabra clave `virtual` y son puras porque el = 0 que sigue a la lista de parámetros especifica que no hay definición para estas funciones en esta clase. *Una clase que contiene al menos una función virtual pura se llama clase abstracta.* La clase Shape contiene dos funciones virtuales puras, `area()` y `scale()`, por lo que definitivamente es una clase abstracta. Veamos un poco más exactamente lo que esto significa.

POLIMORFISMO

CLASES ABSTRACTAS

Aunque tiene una variable miembro, un constructor e incluso una función miembro con una implementación, la clase Shape es una descripción incompleta de un objeto porque las funciones `area()` y `scale()` no están definidas. Por lo tanto, no puede crear instancias de la clase Shape; la clase existe únicamente con el propósito de derivar clases de ella. Como no puede crear objetos de una clase abstracta, no puede pasarlo por valor a una función; un parámetro de tipo Shape no se compilará. De manera similar, no puede devolver un objeto Shape por valor de una función. Sin embargo, los punteros o las referencias a una clase abstracta se pueden usar como parámetros o tipos de retorno, por lo que tipos como `Shape*` y `Shape&` están bien en esta configuración. Es esencial que este sea el caso para obtener un comportamiento polimórfico para los objetos de clase derivados.

POLIMORFISMO

CLASES ABSTRACTAS

Esto plantea la pregunta: "Si no puede crear una instancia de una clase abstracta, ¿por qué la clase abstracta contiene un constructor?" La respuesta es que el constructor de una clase abstracta está ahí para inicializar sus variables miembro. El constructor de una clase abstracta será llamado por un constructor de clase derivada, implícitamente o desde la lista de inicialización del constructor. Si intenta llamar al constructor de una clase abstracta desde cualquier otro lugar, obtendrá un mensaje de error del compilador.

Debido a que el constructor de una clase abstracta generalmente no se puede usar, es una buena idea declararlo como un miembro protegido de la clase, como lo hicimos con la clase Shape. Esto permite llamarlo en la lista de inicialización de un constructor de clases derivadas, pero impide el acceso a él desde cualquier otro lugar.

POLIMORFISMO

CLASES ABSTRACTAS

Tenga en cuenta que un constructor de una clase abstracta no debe llamar a una función virtual pura; el efecto de hacerlo es indefinido. Cualquier clase que se derive de la clase Shape debe definir tanto la función `area()` como la función `scale()`. Si no es así, también es una clase abstracta. Más específicamente, si alguna función virtual pura de una clase base abstracta no está definida en una clase derivada, la función virtual pura se heredará como tal y la clase derivada también será una clase abstracta.

Para ilustrar esto, podría definir una nueva clase llamada `Circle`, que tiene como base la clase `Shape`:

POLIMORFISMO

CLASES ABSTRACTAS

```
// Clase que define un circulo
class Circle : public Shape
{
protected:
    double radius; // Radio del circulo

public:
    Circle(const Point& center, double circleRadius) : Shape{center}, radius{circleRadius} {}

    double area() const override { return radius * radius * PI; }
    void scale(double factor) override { radius *= factor; }
};
```

POLIMORFISMO

CLASES ABSTRACTAS

Las funciones `area()` y `scale()` están definidas, por lo que esta clase no es abstracta. Si alguna de las funciones no estuviera definida, la clase `Circle` sería abstracta. La clase incluye un constructor, que inicializa el subobjeto de la clase base llamando al constructor de la clase base.

Por supuesto, una clase abstracta puede contener funciones virtuales que define y funciones que no son virtuales. Un ejemplo de lo anterior fue la función `move()` en `Shape`. También puede contener cualquier número de funciones virtuales puras.

Veamos un ejemplo de trabajo que usa una clase abstracta. Definiremos una nueva versión de la clase `Box` con la función `volume()` declarada como una función virtual pura. Como clase base polimórfica, por supuesto, también necesita un destructor virtual:

POLIMORFISMO

CLASES ABSTRACTAS

```
class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {}

public:
    virtual ~Box() = default; // Destructor virtual
    virtual double volume() const = 0; // Método para calcular el volumen
};
```

POLIMORFISMO

CLASES ABSTRACTAS

Dado que Box ahora es una clase abstracta, ya no puede crear objetos de este tipo. Esto no habría sido posible incluso si no hubiéramos protegido al constructor. Sin embargo, debido a que estos constructores solo están destinados para su uso en clases derivadas, tiene sentido declararlos protegidos. Las clases Carton y ToughPack en este ejemplo son las mismas que en EJEMPLO 06. Ambos definen la función de `volume()`, por lo que no son abstractos, y podemos usar objetos de estas clases para mostrar que las funciones virtual `volume()` siguen funcionando como antes:

POLIMORFISMO

CLASES ABSTRACTAS

```
// Usando clases abstractas
#include <iostream>
#include "Box.h"
#include "ToughPack.h"
#include "Carton.h"

int main()
{
    ToughPack hardcase {20.0, 30.0, 40.0};
    Carton carton {20.0, 30.0, 40.0, "plastic"};

    Box* pBox =&hardcase; // Puntero Base - direccion derivada
    std::cout << "Volumen de hardcase: " << pBox->volume() << std::endl;

    pBox = &carton; // Nueva direccion derivada
    std::cout << "Volumen de carton: " << pBox->volume() << std::endl;
}
```

EJEMPLO 10

Salida EJEMPLO 10

```
Volumen de hardcase: 20400
Volumen de carton: 22722.4
```

POLIMORFISMO

CLASES ABSTRACTAS

Declarar que `volume()` es una función virtual pura en la clase `Box` garantiza que los miembros de la función `volume()` de las clases `Carton` y `ToughPack` también sean virtuales. Por lo tanto, puede llamarlos a través de un puntero a la clase base y las llamadas se resolverán dinámicamente. El resultado de los objetos `ToughPack` y `Carton` muestra que todo funciona como se esperaba. Los constructores de clases `Carton` y `ToughPack` aún llaman al constructor de clases `Box` que ahora está protegido en sus listas de inicialización.

POLIMORFISMO

Ejercicio 1: Defina una clase base llamada Animal con dos campos: un miembro string para almacenar el nombre del animal (por ejemplo, "Fido") y un miembro entero, peso, que contendrá el peso del Animal en kilos. También incluya un método público, quien(), que devuelve un objeto string que contiene el nombre y el peso del objeto Animal, así como una función virtual pura llamada sonido() que, en una clase derivada, debería devolver una cadena que representa el sonido que hace el animal. Derive al menos tres clases (Oveja, Perro y Vaca) con la clase Animal como base pública e implemente la función sonido() de manera adecuada en cada clase.

POLIMORFISMO

Ejercicio 1: Defina una clase base llamada Animal con dos campos: un miembro string para almacenar el nombre del animal (por ejemplo, "Fido") y un miembro entero, peso, que contendrá el peso del Animal en kilos. También incluya un método público, quien(), que devuelve un objeto string que contiene el nombre y el peso del objeto Animal, así como una función virtual pura llamada sonido() que, en una clase derivada, debería devolver una cadena que representa el sonido que hace el animal. Derive al menos tres clases (Oveja, Perro y Vaca) con la clase Animal como base pública e implemente la función sonido() de manera adecuada en cada clase.

Cree un zoológico de 10 animales utilizando un contenedor std::vector<>. Cree una función regular que reciba una referencia o puntero que permita emitir los sonidos de los 10 animales.

POLIMORFISMO

Ejercicio 2: A partir de la solución del ejercicio 1. Debido a que las vacas son notoriamente conscientes de su peso, el resultado de la función quien() de esta clase ya no debe incluir el peso del animal. Las ovejas, por otro lado, son criaturas caprichosas. tienden a anteponer a su nombre "Woolly", es decir, para una oveja llamada "Pete" quien() debería devolver una cadena que contenga "Woolly Pete". Además de eso, también debería reflejar el peso real de una Oveja, que es su peso total (como se almacena en el objeto base Animal) menos el de su lana (conocido por la propia Oveja). digamos que la lana de una oveja por defecto pesa el 10 por ciento de su peso total.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Una excepción es un objeto temporal, de cualquier tipo, que se utiliza para señalar un error. En teoría, una excepción puede ser de un tipo fundamental, como int o const char*, pero por lo general y más apropiadamente es un objeto de un tipo de clase.

El propósito de un objeto de excepción es transportar información desde el punto en el que ocurrió el error hasta el código que debe manejar el error. En muchas situaciones, se involucra más de una pieza de información, por lo que es mejor hacerlo con un objeto de un tipo de clase. Cuando reconoce que algo salió mal en el código, puede señalar el error lanzando una excepción. El término arrojar efectivamente indica lo que sucede. El objeto de excepción se lanza a otro bloque de código que detecta la excepción y la trata. El código que puede generar excepciones debe estar dentro de un bloque especial llamado bloque try si se va a capturar una excepción.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Si una sentencia que no está dentro de un bloque try arroja una excepción o una sentencia dentro de un bloque try arroja una excepción que no se detecta, el programa finaliza. Un bloque try es seguido inmediatamente por uno o más bloques catch. Cada bloque catch contiene código para manejar un tipo particular de excepción; por este motivo, a veces se hace referencia a un bloque catch como un controlador de excepciones. Todo el código que se ocupa de los errores que provocan el lanzamiento de excepciones está dentro de bloques catch que están completamente separados del código que se ejecuta cuando todo funciona como debería.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

El código que puede lanzar excepciones debe estar dentro de un código try

Cuando se lanza una excepción, los bloques catch se examinan en secuencia. El control se transfiere al primer bloque catch que tiene un parámetro que coincide con el tipo de excepción, mediante conversión implícita de la excepción si es necesario.

```
try
{
    // Code that may throw exceptions...
}

catch (parameter specifying exception type 1)
{
    // Code to handle the exception...
}

catch (parameter specifying exception type 2)
{
    // Code to handle the exception...
}

...
catch (parameter specifying exception type n)
{
    // Code to handle the exception...
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

El código en un bloque catch se ejecuta solo cuando se lanza una excepción de un tipo coincidente. Si un bloque de prueba no arroja una excepción, entonces no se ejecuta ninguno de los bloques de captura que siguen al bloque de prueba. Un bloque de prueba siempre se ejecuta comenzando con la primera declaración que sigue a la llave de apertura.

Lanzando una excepción

Veamos un ejemplo de excepción. Aunque siempre debe usar objetos de clase para las excepciones comenzaremos usando tipos básicos porque esto mantendrá el código simple mientras explicamos lo que sucede.

Lanzas una excepción usando una expresión throw, que escribes usando la palabra clave throw. Aquí hay un ejemplo:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

```
try
{
    if (test > 5)
        throw "test es mayor que 5"; // Lanza una excepción de tipo const char*
    // Código que se ejecuta si no se produce una excepción
}
catch (const char* message)
{
    // Código para procesar excepción
    // ...que se ejecuta si se lanza una excepción de tipo 'char*' o 'const char*'
    std::cout << message << std::endl;
}
```

Si el valor de prueba es mayor que 5, la sentencia `throw` lanza una excepción. En este caso, la excepción es el literal “`test es mayor que 5`”. El control se transfiere inmediatamente fuera del bloque `try` al primer controlador para el tipo de excepción que se lanzó: `const char*`. Solo hay un controlador aquí, que detecta excepciones de tipo `const char*`, por lo que se ejecuta la declaración en el bloque `catch`, y esto muestra la excepción.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

El compilador ignora la palabra clave `const` al hacer coincidir el tipo de una excepción que se lanzó con el tipo de parámetro `catch`.

Veamos un segundo ejemplo de utilización con tipos fundamentales de la estructura `try-catch`.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

```
#include <iostream>

int main()
{
    for (size_t i {}; i < 7; ++i)
    {
        try
        {
            if (i < 3)
                throw i;

            std::cout << "Valor de i: " << i << std::endl;

            if (i > 5)
                throw "Valor ilegal..excepcion > 5";

            std::cout << "Fin de bloque try." << std::endl;
        }
        catch (size_t i) // Atrapa excepciones de tipo size_t
        {
            std::cout << "Excepción capturada, i: " << i << std::endl;
        }
        catch (const char* message) // Atrapa excepciones de tipo char*
        {
            std::cout << "Excepción capturada - valor: \" " << message << " \" << std::endl;
        }

        std::cout << "Fin del cuerpo for" << " Valor de i:" << i << std::endl;
    }
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

El proceso de manejo de excepciones

```
try
{
    ...
    throw theException;
    ...
}
```

1- La expresión `throw` es usada para inicializar un objeto temporal, `temp` como:

```
ExceptionType temp {theException};
```

```
catch(Type1 ex)
{
    ...
}
```

4- La copia de la excepción se usa para inicializar el parámetro como:

```
TypeN es {temp};
Y el control se transfiere al handler.
```

2- Los destructores son llamados para todos los objetos automáticos definidos en el bloque `try`.

```
...
catch(TypeN ex)
{
    ...
}
catch(TypeLast ex)
{
    ...
}
```

3- El primer handler cuyo tipo de parámetro coincide con la excepción es seleccionado.

5- Cuando el handler finaliza, a menos que el código del controlador determine lo contrario, la ejecución continúa con la instrucción que sigue al último handler del bloque `try`.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

El proceso de manejo de excepciones

Un bloque try es un bloque de instrucciones y sabe que un bloque de instrucciones define un alcance. Lanzar una excepción deja el bloque try inmediatamente, por lo que en ese momento se destruyen todos los objetos automáticos que se han definido dentro del bloque try antes de que se lance la excepción. El hecho de que ninguno de los objetos automáticos creados en el bloque try exista en el momento en que se ejecuta el código del controlador es muy importante; implica que no debe lanzar un objeto de excepción que sea un puntero a un objeto que sea local para el bloque try. También es la razón por la cual el objeto de excepción se copia en el proceso de lanzamiento.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

El proceso de manejo de excepciones

Debido a que la expresión `throw` se usa para inicializar un objeto temporal y, por lo tanto, crea una copia de la excepción, puede lanzar objetos que sean locales para el bloque `try` pero no punteros a objetos locales. La copia del objeto se usa para inicializar el parámetro para el bloque `catch` que se selecciona para manejar la excepción. Un bloque `catch` también es un bloque de instrucciones, por lo que cuando un bloque `catch` ha terminado de ejecutarse, todos los objetos automáticos que son locales para él (incluido el parámetro) se destruirán.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

El proceso de manejo de excepciones

A menos que transfiera el control fuera del bloque catch usando, por ejemplo, una instrucción de retorno, la ejecución continúa con la declaración inmediatamente. Siguiendo el último bloque catch para el bloque try. Una vez que se ha seleccionado un controlador para una excepción y se le ha pasado el control, la excepción se considera manejada. Esto es cierto incluso si el bloque catch está vacío y no hace nada.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

A menos que transfiera el control fuera del bloque catch usando, por ejemplo, una instrucción de retorno, la ejecución continúa con la declaración inmediatamente. Siguiendo el último bloque catch para el bloque try. Una vez que se ha seleccionado un controlador para una excepción y se le ha pasado el control, la excepción se considera manejada. Esto es cierto incluso si el bloque catch está vacío y no hace nada.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

Los bloques try encierran código que puede generar una excepción. Sin embargo, esto no significa que el código que genera una excepción deba estar físicamente entre las llaves que limitan el bloque de prueba. Solo necesita estar lógicamente dentro del bloque try. Si se llama a una función dentro de un bloque try , cualquier excepción que se genere y no se capture dentro de esa función puede ser capturada por uno de los bloques de captura para el bloque try. La siguiente imagen lo ilustra. Se muestran dos llamadas de función dentro del bloque try: fun1() y fun2(). Las excepciones de tipo ExceptionType que se lanzan dentro de cualquiera de las funciones pueden ser capturadas por el bloque catch que sigue al bloque try.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

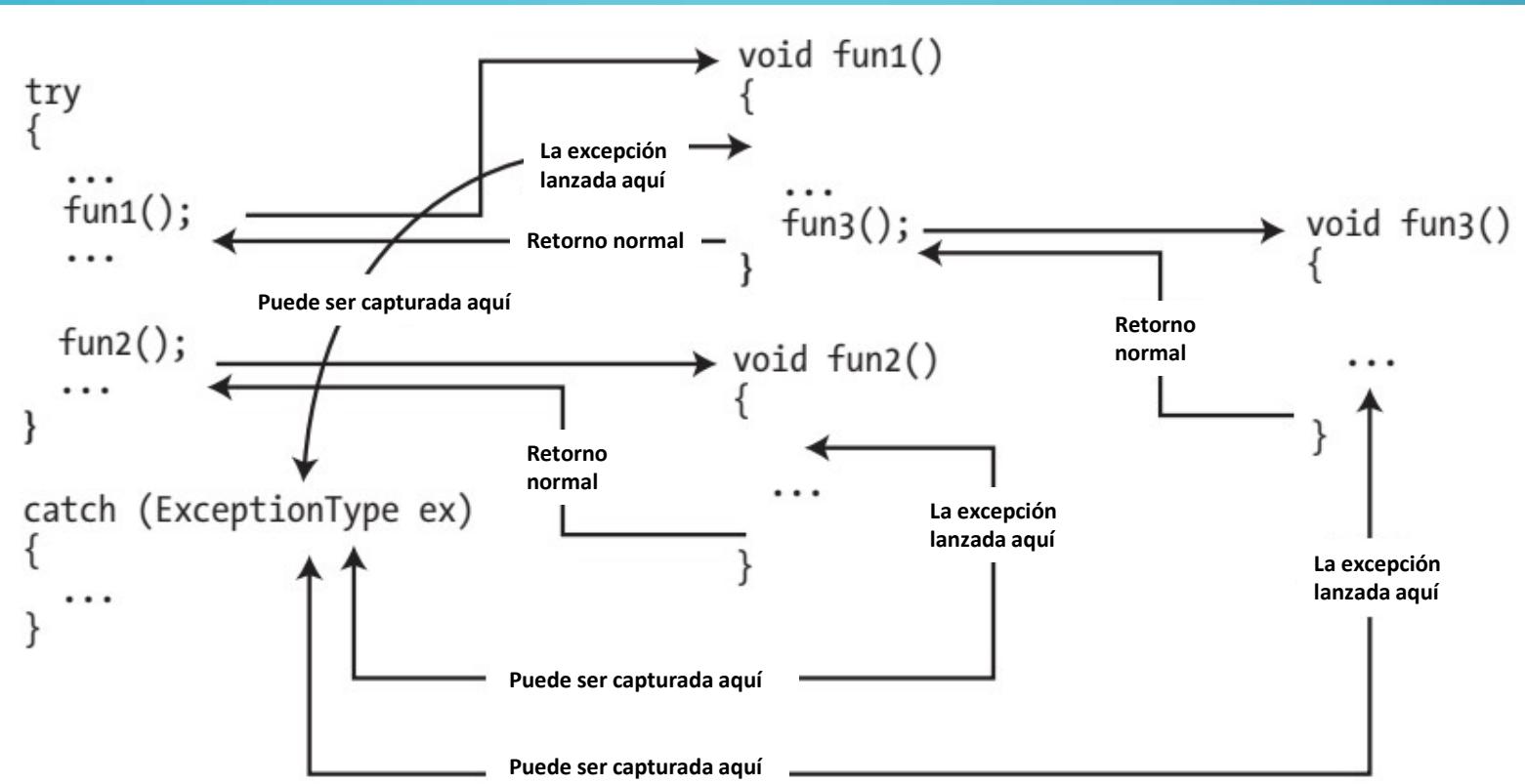
Código que hace que se lance una excepción

Una excepción que se lanza pero no se captura dentro de una función puede pasarse a la función de llamada del siguiente nivel. Si no se atrapa allí, se puede pasar al siguiente nivel; esto se ilustra en la siguiente figura con la excepción lanzada en fun3() cuando es llamado por fun1(). No hay un bloque de prueba en fun1(), por lo que las excepciones lanzadas por fun3() se pasarán a la función que llamó a fun1(). Si una excepción alcanza un nivel en el que no existe más controlador de captura y aún no se detecta, entonces el programa generalmente finaliza.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

Excepción lanzada por funciones llamadas dentro de un bloque try



ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

Por supuesto, si se llama a la misma función desde diferentes puntos de un programa, las excepciones que puede generar el código en el cuerpo de la función pueden ser manejadas por diferentes bloques catch en diferentes momentos.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

Llamando a la misma función desde diferentes bloques de prueba

```
try
{
    ...
    fun1(); ←
}
catch (ExceptionType ex)
{
    ...
}

try
{
    fun1(); ←
}
catch (ExceptionType ex)
{
    ...
}
```

Mientras se ejecuta esta llamada
el código en la función fun1() está
lógicamente dentro del bloque try
superior

Mientras se ejecuta esta llamada
el código en la función fun1() está
lógicamente dentro del bloque try
inferior

```
void fun1()
{
    ...
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

Como consecuencia de la llamada en el primer bloque try, el bloque catch para ese bloque de prueba maneja cualquier excepción de tipo ExceptionType lanzada por fun1(). Cuando se llama a fun1() en el segundo bloque try, el controlador catch para ese bloque try se ocupa de cualquier excepción de tipo ExceptionType que se produzca. A partir de esto, debería poder ver que puede elegir manejar las excepciones en el nivel que sea más conveniente para la estructura y operación de su programa. En un caso extremo, podría capturar todas las excepciones que surgieron en cualquier parte de un programa en main() simplemente encerrando el código en main() en un bloque try y agregando una variedad adecuada de bloques de captura

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Bloques try anidados

```
try
{ // bloque try exterior
...
try
{ // bloque try interior
...
}
catch (ExceptionType1 ex)
{
...
}
...
}
catch (ExceptionType2 ex)
{
...
}
```

Este handler detecta las excepciones ExceptionType1 lanzadas en el bloque try interno.

Este handler detecta las excepciones ExceptionType2 lanzadas en el bloque try externo, así como las excepciones no detectadas de ese tipo desde el bloque try interno.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

La figura muestra un controlador para cada bloque try, pero en general puede haber varios. Los bloques catch capturan excepciones de diferentes tipos, pero podrían capturar excepciones del mismo tipo. Cuando el código en un bloque try interno arroja una excepción, sus controladores tienen la primera oportunidad de manejarlo. Cada controlador para el bloque de prueba se verifica en busca de un tipo de parámetro coincidente y, si ninguno coincide, los controladores para el bloque try externo tienen la posibilidad de detectar la excepción. Puede anidar bloques try de esta manera a cualquier profundidad que sea apropiada para su aplicación.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

Cuando el código lanza una excepción en el bloque try externo, los controladores catch de ese bloque la manejan, incluso si la sentencia que origina la excepción precede al bloque try interno. Los controladores catch para el bloque de prueba interno nunca pueden estar involucrados en el tratamiento de las excepciones lanzadas por el código dentro del bloque de prueba externo. El código dentro de ambos bloques try puede llamar a funciones, en cuyo caso el código dentro del cuerpo de la función está lógicamente dentro del bloque try que lo llamó.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

Cualquiera o todo el código dentro del cuerpo de la función también podría estar dentro de su propio bloque try, en cuyo caso este bloque try estaría anidado dentro del bloque try que llamó a la función. Todo suena bastante complicado en palabras, pero es mucho más fácil en la práctica. Reuniremos un ejemplo simple en el que se lanza una excepción y luego veremos dónde termina. Para simplificar que lanzaremos excepciones de tipo int y tipo long en lugar de objetos de un tipo de clase. Ingrese al repositorio y descargue el código que se encuentra en la carpeta Excepciones/Ejemplo1 (Clase11_ExcepcionesAnid).

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Código que hace que se lance una excepción

Aunque estos ejemplos muestran claramente la mecánica de lanzar y capturar excepciones y lo que sucede con los bloques de prueba anidados, las excepciones que usan no son particularmente realistas. Las excepciones en los programas reales son invariablemente objetos de clase. Por lo tanto, pasemos a echar un vistazo más de cerca a las excepciones que son objetos.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Objetos de clase como excepciones

Puede lanzar cualquier tipo de objeto de clase como una excepción. Sin embargo, tenga en cuenta que la idea de un objeto de excepción es comunicar información al controlador sobre lo que salió mal. Por lo tanto, suele ser apropiado definir una clase de excepción específica diseñada para representar un tipo particular de problema. Es probable que esto sea específico de la aplicación, pero sus objetos de clase de excepción casi siempre contienen algún tipo de mensaje que explica el problema y posiblemente algún tipo de código de error. También puede organizar un objeto de excepción para proporcionar información adicional sobre el origen del error en cualquier forma que sea adecuada. Definamos una clase de excepción simple. Lo haremos en un archivo de encabezado con un nombre bastante genérico, MyTroubles.h:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Objetos de clase como excepciones

```
// Definición de clase de excepción
#ifndef MYTROUBLES_H_
#define MYTROUBLES_H_

#include <string>
#include <string_view>

class Trouble
{
private:
    std::string message;

public:
    Trouble(std::string_view str = "Hay un problema") : message {str} {}

    std::string_view what() const { return message; }
};

#endif /* MYTROUBLES_H_ */
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Objetos de clase como excepciones

Los objetos de la clase Trouble simplemente almacenan un mensaje que indica un problema y, por lo tanto, son ideales como objetos de excepción simples. Se define un mensaje predeterminado en la lista de parámetros para el constructor, por lo que puede usar el constructor predeterminado para obtener un objeto que contenga el mensaje predeterminado. Por supuesto, si debe usar dichos mensajes predeterminados es otra cuestión completamente diferente. Recuerde, la idea suele ser proporcionar información sobre la causa del problema para ayudar con el diagnóstico del problema. La función miembro what() devuelve el mensaje actual. Para mantener manejable la lógica del manejo de excepciones, sus funciones deben garantizar que las funciones miembro de una clase de excepción no generen excepciones.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Objetos de clase como excepciones

Más adelante, verá cómo puede señalar explícitamente que una función miembro nunca generará excepciones.

Averigüemos qué sucede cuando se lanza un objeto de clase lanzando algunos. Como en los ejemplos anteriores, no nos molestaremos en crear errores. Simplemente lanzaremos objetos de excepción para que pueda seguir lo que les sucede en diversas circunstancias. Ejercitaremos la clase de excepción con un ejemplo simple que arroja algunos objetos de excepción en un bucle:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

```
// Ejemplo03. Lanza un objeto de excepcion
#include <iostream>
#include "MyTroubles.h"

void trySomething(int i);

int main()
{
    for (int i {}; i < 2; ++i)
    {
        try
        {
            trySomething(i);
        }
        catch (const Trouble& t)
        {
            // Cual sera el problema?
            std::cout << "Excepcion: " << t.what() << std::endl;
        }
    }
}

void trySomething(int i)
{
    if (i == 0)
        throw Trouble {};
    else
        throw Trouble {"Nadie sabe el problema que he visto..."};
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Objetos de clase como excepciones

El parámetro para el bloque catch es una referencia. Recuerde que un objeto de excepción siempre se copia cuando se lanza, por lo que si no especifica el parámetro para un bloque catch como referencia, se copiará por segunda vez, sin necesidad. La secuencia de eventos cuando se lanza un objeto de excepción es que primero el objeto se copia para crear un objeto temporal y el original se destruye porque se sale del bloque de prueba y el objeto queda fuera del alcance. La copia se pasa al controlador catch, por referencia si el parámetro es una referencia. Si desea observar cómo ocurren estos eventos, simplemente agregue un constructor de copia y un destructor que contenga algunas declaraciones de salida a la clase Problema.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Coincidencia de un controlador de captura con una excepción

Anteriormente dijimos que los controladores que siguen a un bloque try se examinan en la secuencia en la que aparecen en el código, y se ejecutará el primer controlador cuyo tipo de parámetro coincide con el tipo de la excepción. Con las excepciones que son tipos básicos (en lugar de tipos de clase), es necesaria una coincidencia de tipo exacta con el parámetro en el bloque catch. Con las excepciones que son objetos de clase, se pueden aplicar conversiones implícitas para hacer coincidir el tipo de excepción con el tipo de parámetro de un controlador. Cuando el tipo de parámetro se compara con el tipo de excepción que se lanzó, lo siguiente se considera una coincidencia:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Coincidencia de un controlador de captura con una excepción

- El tipo de parámetro es el mismo que el tipo de excepción, ignorando const.
- El tipo del parámetro es una clase base directa o indirecta del tipo de clase de excepción, o una referencia a una clase base directa o indirecta de la clase de excepción, ignorando const.
- La excepción y el parámetro son punteros, y el tipo de excepción se puede convertir implícitamente al tipo de parámetro, ignorando const.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Coincidencia de un controlador de captura con una excepción

Las posibles conversiones de tipo enumeradas aquí tienen implicaciones sobre cómo se secuencia los bloques catch para un bloque try. Si tiene varios controladores para tipos de excepción dentro de la misma jerarquía de clases, entonces el tipo de clase más derivado debe aparecer primero y el tipo de clase más básico al final. Si un controlador para un tipo base aparece antes que un controlador para un tipo derivado de esa base, entonces el tipo base siempre se selecciona para controlar las excepciones de clases derivadas. En otras palabras, el controlador del tipo derivado nunca se ejecuta. Ejemplo:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

```
// Ejemplo04 - MyTroubles.h Clases excepción
#ifndef MYTROUBLES_H
#define MYTROUBLES_H
#include <string>
#include <string_view>

class Trouble
{
private:
    std::string message;
public:
    Trouble(std::string_view str = "Hay un problema") : message {str} {}
    virtual ~Trouble() = default; // Las clases Base deben tener siempre un destructor virtual!
    virtual std::string_view what() const { return message; }
};

// Clase de excepción derivada
class MoreTrouble : public Trouble
{
public:
    MoreTrouble(std::string_view str = "Hay mas problemas...") : Trouble {str} {}
};

// Clase de excepción derivada
class BigTrouble : public MoreTrouble
{
public:
    BigTrouble(std::string_view str = "Realmente un gran problema...") : MoreTrouble {str} {}
};
#endif
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Coincidencia de un controlador de captura con una excepción

Tenga en cuenta que el miembro `what()` y el destructor de la clase base se han declarado como virtuales. Por lo tanto, la función `what()` también es virtual en las clases derivadas de `Trouble`. No hace mucha diferencia aquí, pero en principio permitiría que las clases derivadas redefinieran `what()`. Sin embargo, es importante recordar declarar un destructor virtual en una clase base. Aparte de las diferentes cadenas predeterminadas para el mensaje, las clases derivadas no agregan nada a la clase base. A menudo, el simple hecho de tener un nombre de clase diferente puede diferenciar un tipo de problema de otro. Simplemente lance una excepción de un tipo particular cuando surge ese tipo de problema; las partes internas de las clases no tienen que ser diferentes. El uso de un bloque `catch` diferente para capturar cada tipo de clase proporciona los medios para distinguir diferentes problemas. Este es el código para generar excepciones de los tipos `Trouble`, `MoreTrouble` y `BigTrouble`, así como los controladores para detectarlas:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Matching a Catch Handler to an Exception

```
// Lanzar y atrapar objetos en una jerarquía de clases
#include <iostream>
#include "MyTroubles.h"

int main()
{
    Trouble trouble;
    MoreTrouble moreTrouble;
    BigTrouble bigTrouble;

    for (int i {}; i < 7; ++i)
    {
        try
        {
            if (i == 3)
                throw trouble;
            else if (i == 5)
                throw moreTrouble;
            else if (i == 6)
                throw bigTrouble;
        }
    }
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Coincidencia de un controlador de captura con una excepción

```
catch (const BigTrouble& t)// Clase derivada  
de MoreTrouble  
{  
    std::cout << "Objeto BigTrouble atrapado:  
" << t.what() << std::endl;  
}  
catch (const MoreTrouble& t)// Clase derivada  
de Trouble  
{  
    std::cout << "Objeto MoreTrouble object  
atrapado: " << t.what() << std::endl;  
}  
catch (const Trouble& t)// Clase Base  
{  
    std::cout << "Objeto Trouble atrapado: "  
<< t.what() << std::endl;  
}  
    std::cout << "Fin de the for loop (despues  
de los bloques catch) - i es " << i <<  
std::endl;  
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Captura de excepciones de clases derivadas con un controlador de clase base

Las excepciones de los tipos de clase derivados se convierten implícitamente en un tipo de clase base con el fin de hacer coincidir un parámetro de bloque catch, por lo que podría capturar todas las excepciones lanzadas en el ejemplo anterior con un solo controlador. Puede modificar el ejemplo anterior para ver que esto suceda. Simplemente elimine o comente los dos controladores de clase derivados de main() en el ejemplo anterior:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Captura de excepciones de clases derivadas con un controlador de clase base

```
// Atrapar excepciones con un controlador de clase base
#include <iostream>
#include "MyTroubles.h"

int main()
{
    Trouble trouble;
    MoreTrouble moreTrouble;
    BigTrouble bigTrouble;

    for (int i {}; i < 7; ++i)
    {
        try
        {
            if (i == 3)
                throw trouble;
            else if (i == 5)
                throw moreTrouble;
            else if (i == 6)
                throw bigTrouble;
        }
        catch (const Trouble& t)
        {
            std::cout << "Objeto Trouble object atrapado: " << t.what() << std::endl;
        }
        std::cout << "Fin de for loop (despues de los bloques catch) - i es " << i << std::endl;
    }
}
```

EJEMPLO 05

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Captura de excepciones de clases derivadas con un controlador de clase base

Salida EJEMPLO05

```
Fin de for loop (despues de los bloques catch) - i es 0
Fin de for loop (despues de los bloques catch) - i es 1
Fin de for loop (despues de los bloques catch) - i es 2
Objeto Trouble object atrapado: Hay un problema
Fin de for loop (despues de los bloques catch) - i es 3
Fin de for loop (despues de los bloques catch) - i es 4
Objeto Trouble object atrapado: Hay mas problemas...
Fin de for loop (despues de los bloques catch) - i es 5
Objeto Trouble object atrapado: Realmente un gran problema...
Fin de for loop (despues de los bloques catch) - i es 6
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Captura de excepciones de clases derivadas con un controlador de clase base

El bloque `catch` con el parámetro de tipo `const Trouble&` ahora captura todas las excepciones lanzadas en el bloque `try`. Si el parámetro en un bloque `catch` es una referencia a una clase base, entonces coincide con cualquier excepción de clase derivada. Entonces, aunque la salida proclama "Objeto Trouble capturado" para cada excepción, la salida en realidad corresponde a objetos de otras clases que se derivan de `Trouble`. El tipo dinámico se conserva cuando la excepción se pasa por referencia. Para verificar que esto sea así, puede obtener el tipo dinámico y mostrarlo usando el operador `typeid()`. Simplemente modifique el código para el controlador a lo siguiente:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Captura de excepciones de clases derivadas con un controlador de clase base

```
catch (const Trouble& t)
{
    std::cout << typeid(t).name() << " object caught: " << t.what() << std::endl;
```

El operador `typeid()` devuelve un objeto de la clase `type_info`, y llamar a su miembro `name()` devuelve el nombre de la clase. Con esta modificación del código, el resultado muestra que las excepciones de clases derivadas aún conservan sus tipos dinámicos, aunque la referencia en el controlador de excepciones sea a la clase base. La salida de esta versión del programa se ve así:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Relanzamiento de excepciones

Cuando un controlador detecta una excepción, puede volver a lanzarla para permitir que un controlador de un bloque try externo la capture. Para volver a lanzar la excepción actual debemos utilizar la palabra clave `throw`:

```
throw; // Relanza la excepcion
```

Esto vuelve a lanzar el objeto de excepción existente sin copiarlo. Puede volver a generar una excepción si un controlador que detecta excepciones de más de un tipo de clase derivada descubre que el tipo de excepción requiere que se pase a otro nivel del bloque de prueba. Es posible que desee registrar (por ej logging) el punto en el programa donde se lanzó una excepción, antes de volver a lanzarla.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Relanzamiento de excepciones

O puede necesitar limpiar algunos recursos (liberar algo de memoria, cerrar una conexión de base de datos, etc.) antes de volver a generar la excepción para manejara. Tenga en cuenta que volver a lanzar una excepción desde el bloque de prueba interno no hace que la excepción esté disponible para otros controladores para el bloque de prueba interno. Cuando se ejecuta un controlador, cualquier excepción que se genere (incluida la excepción actual) debe ser capturada por un controlador para un bloque de prueba que incluya el controlador actual, como se ilustra en la siguiente figura. El hecho de que una excepción que se vuelve a generar no se copie es importante, especialmente cuando la excepción es un objeto de clase derivada que inicializó un parámetro de referencia de clase base. Demostraremos esto con un ejemplo.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Relanzamiento de excepciones

```
try      // Outer try block
{
    ...
    try // Inner try block
    {
        if(...)
            throw ex;
        ...
    } catch (ExType& ex)
    {
        ...
        throw; ←
    } catch (AType& ex)
    {
        ...
    }
    catch (ExType& ex)
    {
        // Handle ex...
    }
}
```

Este controlador captura la excepción ex que es lanzada en el bloque try interno

Esta sentencia relanza ex sin copiarla para que pueda ser capturada por un controlador para el bloque try externo.

Este controlador captura la excepción ex que fue relanzada en el bloque try interno

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Relanzamiento de excepciones

```
// Relanzamiento de excepciones
#include <iostream>
#include "MyTroubles.h"

int main()
{
    Trouble trouble;
    MoreTrouble moreTrouble;
    BigTrouble bigTrouble;

    for (int i {}; i < 7; ++i)
    {
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

EJEMPLO 06

```
try
{
    try
    {
        if (i == 3)
            throw trouble;
        else if (i == 5)
            throw moreTrouble;
        else if (i == 6)
            throw bigTrouble;
    }
    catch (const Trouble& t)
    {
        if (typeid(t) == typeid(Trouble))
            std::cout << "Objeto Trouble capturado
en bloque interno: " << t.what() << std::endl;
        else
            throw; // Relanzar excepción actual
    }
    catch (const Trouble& t)
    {
        std::cout << typeid(t).name() << " object
capturado en bloque externo: "
                << t.what() << std::endl;
    }
    std::cout << "Fin de for loop (despues de los bloques
catch) - i es " << i << std::endl;
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Relanzamiento de excepciones

Salida EJEMPLO 06

```
Fin de for loop (despues de los bloques catch) - i es 0
Fin de for loop (despues de los bloques catch) - i es 1
Fin de for loop (despues de los bloques catch) - i es 2
Objeto Trouble capturado en bloque interno: Hay un problema
Fin de for loop (despues de los bloques catch) - i es 3
Fin de for loop (despues de los bloques catch) - i es 4
11MoreTrouble object capturado en bloque externo: Hay mas problemas...
Fin de for loop (despues de los bloques catch) - i es 5
10BigTrouble object capturado en bloque externo: Realmente un gran problema...
Fin de for loop (despues de los bloques catch) - i es 6
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Relanzamiento de excepciones

El bucle for funciona como en el ejemplo anterior, pero esta vez hay un bloque try anidado dentro de otro. La misma secuencia de objetos excepción que los objetos del ejemplo anterior es lanzada en el bloque try interno, y todos los objetos de excepción son capturados por el bloque de captura coincidente porque el parámetro es una referencia a la clase base, Trouble. La sentencia if en el bloque catch prueba el tipo de clase del objeto pasado y ejecuta la sentencia de salida si es del tipo Trouble. Para cualquier otro tipo de excepción, la excepción se vuelve a generar y, por lo tanto, está disponible para que la capture el bloque catch del bloque try externo. El parámetro también es una referencia a Trouble, por lo que captura todos los objetos de clase derivados. La salida muestra que atrapa los objetos vueltos a lanzar y todavía están en perfectas condiciones.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Relanzamiento de excepciones

Puede pensar que la declaración `throw` en el controlador del bloque `try` interno es equivalente a la siguiente sentencia:

```
throw t; // Relanza excepción
```

Después de todo, solo está volviendo a lanzar la excepción, ¿no es así? La respuesta es no; hay una diferencia crucial. Si realiza esta modificación en el código del programa y lo vuelve a ejecutar, obtendrá el siguiente resultado:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Relanzamiento de excepciones

```
Fin de for loop (despues de los bloques catch) - i es 0
Fin de for loop (despues de los bloques catch) - i es 1
Fin de for loop (despues de los bloques catch) - i es 2
Objeto Trouble capturado en bloque interno: Hay un problema
Fin de for loop (despues de los bloques catch) - i es 3
Fin de for loop (despues de los bloques catch) - i es 4
7Trouble object capturado en bloque externo: Hay mas problemas...
Fin de for loop (despues de los bloques catch) - i es 5
7Trouble object capturado en bloque externo: Realmente un gran problema...
Fin de for loop (despues de los bloques catch) - i es 6
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Relanzamiento de excepciones

La sentencia con un objeto de excepción explícito especificado genera una nueva excepción, no vuelve a generar la original. Esto da como resultado que se copie el objeto de excepción original, usando el constructor de copia para la clase Trouble. Se produce el recorte del objeto (object slicing). La parte derivada de cada objeto se corta, por lo que solo queda el subobjeto de la clase base en cada caso. Puede ver en el resultado que el operador typeid() identifica todas las excepciones como tipo Trouble. Por lo tanto:

Lanzar siempre por valor, atrapar por referencia y volver a lanzar usando throw; declaración.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Excepciones y fugas de recursos

Asegurarse de que se detecten todas las excepciones evita fallas catastróficas del programa. Y atraparlos con bloques de captura adecuadamente posicionados y lo suficientemente finos le permite manejar adecuadamente todos los errores. El resultado es un programa que, en todo momento, presenta el resultado deseado o puede informar al usuario con precisión qué salió mal. ¡Pero este no es el final de la historia! Un programa que parece estar funcionando sólidamente desde el exterior aún puede contener defectos ocultos. Considere, por ejemplo, el siguiente programa:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Excepciones y fugas de recursos

```
// EJEMPLO 08. Excepciones que resultan en pérdidas de recursos.

#include <iostream>
#include <cmath>
#include "MyTroubles.h"

double DoComputeValue(double); // Función que calcula un valor simple
double* ComputeValues(size_t howMany); // Función que realiza un calculo sobre un array

int main()
{
    try
    {
        double* values = ComputeValues(10000);
        // desafortunadamente, no llegaremos tan lejos..
        delete[] values;
    }
    catch (const Trouble&)
    {
        std::cout << "Excepción capturada!" << std::endl;
    }
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Excepciones y fugas de recursos

```
double* ComputeValues(size_t howMany)
{
    double* values = new double[howMany];

    for (size_t i = 0; i < howMany; ++i)
        values[i] = DoComputeValue(i);

    return values;
}

double DoComputeValue(double value)
{
    if (value < 100)
        return std::sqrt(value); // Retorna raiz cuadrada de value
    else
        throw Trouble{"Problema de computo!!"};
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Excepciones y fugas de recursos

Si ejecuta este programa, se lanza una excepción Trouble tan pronto como el contador de bucles en `ComputeValues()` llega a 100. Debido a que la excepción se detecta en `main()`, el programa no falla. Incluso asegura al usuario que todo está bien. Si se tratara de un programa real, incluso podría informar al usuario sobre qué salió mal exactamente con esta operación y permitirle continuar. ¡Pero eso no significa que esté fuera de peligro! ¿Puedes identificar qué más ha fallado con este programa? La función `ComputeValues()` asigna un array de valores dobles en el almacenamiento libre, intenta llenarlos y luego devuelve el array a su llamador. Es responsabilidad de quien llama, en este caso `main()`, desasignar esta memoria. Sin embargo, debido a que se lanza una excepción a la mitad de la ejecución de `ComputeValues()`, su array `values` nunca se devuelve a `main()`. Por lo tanto, la matriz tampoco se desasigna nunca.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Excepciones y fugas de recursos

En otras palabras, jacobamos de fugar una matriz de 10000 doubles! Suponiendo que DoComputeValue() lleve inherentemente a la excepción de Trouble ocasional, el único lugar donde podemos corregir esta fuga es en la función ComputeValues(). Después de todo, main() ni siquiera recibe un puntero a la memoria fugada, por lo que es poco lo que se puede hacer al respecto. Una primera solución obvia sería agregar un bloque try a ComputeValues() de la siguiente manera:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Excepciones y fugas de recursos

```
double* ComputeValues(size_t howMany)
{
    double* values = new double[howMany];

    try
    {
        for (size_t i = 0; i < howMany; ++i)
            values[i] = DoComputeValue(i);

        return values;
    }
    catch (const Trouble&)
    {
        std::cout << "Problema detectado... Liberando memoria..." << std::endl;
        delete[] values;
        throw;
    }
}
```

Salida

Problema detectado... Liberando memoria...
Excepción capturada!

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

RAII (Adquisición de Recursos es Inicialización)

Uno de los sellos distintivos del C++ moderno es la técnica RAII, abreviatura de "la adquisición de recursos es inicialización". Su premisa es que cada vez que adquiere un recurso debe hacerlo inicializando un objeto. La memoria en el almacenamiento libre es un recurso, pero otros ejemplos incluyen identificadores de archivos (mientras los mantienen, es posible que otros procesos a menudo no accedan a un archivo), mutexes (utilizados para la sincronización de subprocessos), conexiones de red , etc. Según RAII, cada recurso de este tipo debe ser administrado por un objeto, ya sea asignado en la pila o como una variable miembro (campo). El truco para evitar fugas de recursos es entonces que, por defecto, el destructor de ese objeto se asegura de que el recurso esté siempre liberado. Vamos a crear una clase RAII simple para demostrar cómo funciona este mecanismo:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

RAII (Adquisición de Recursos es Inicialización)

```
class DoubleArrayRAII final
{
private:
    double* resource;
public:
    DoubleArrayRAII(size_t size) : resource{ new double[size] } {}
    ~DoubleArrayRAII()
    {
        std::cout << "Liberando memoria..." << std::endl;
        delete[] resource;
    }

    // Borrar constructor de copia y operador de asignación.
    DoubleArrayRAII(const DoubleArrayRAII&) = delete;
    DoubleArrayRAII& operator=(const DoubleArrayRAII&) = delete;
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

RAII (Adquisición de Recursos es Inicialización)

```
// Operador de subíndice de array
double& operator[](size_t index) noexcept { return resource[index]; }
const double& operator[](size_t index) const noexcept { return resource[index]; }

// Función para acceder al recurso encapsulado
double* get() const noexcept { return resource; }

// Función para indicar al objeto RAII que entregue el recurso.
// Una vez llamado, el objeto RAII ya no liberará el recurso tras
// la destrucción. Devuelve el recurso en el proceso.
double* release() noexcept
{
    double* result = resource;
    resource = nullptr;
    return result;
};
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

RAll (Adquisición de Recursos es Inicialización)

El recurso, en este caso la memoria para contener una matriz doble, es adquirido por el constructor del objeto RAll y liberado por su destructor. Para esta clase RAll, es fundamental que el recurso, la memoria asignada para su matriz, se libere solo una vez. Esto implica que no debemos permitir que se hagan copias de un DoubleArrayRAll existente; de lo contrario, terminaremos teniendo dos objetos DoubleArrayRAll apuntando al mismo recurso. Esto se logra eliminando ambos miembros de copia (el operador de asignación también puede usarse para copiar un objeto).

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

RAlI (Adquisición de Recursos es Inicialización)

Un objeto RAlI a menudo imita el recurso que administra al agregar las funciones y los operadores de miembros apropiados. En nuestro caso, el recurso es un array, por lo que definimos los operadores de subíndice de array familiares. Además de estas, normalmente existen funciones adicionales para acceder al recurso en sí (nuestra función `get()`) y, a menudo, también para liberar al objeto RAlI de su responsabilidad de liberar el recurso (la función `release()`). Con la ayuda de esta clase RAlI, puede reescribir con seguridad la función `ComputeValues()` de la siguiente manera:

```
double* ComputeValues(size_t howMany)
{
    DoubleArrayRAII values{howMany};

    for (size_t i = 0; i < howMany; ++i)
        values[i] = DoComputeValue(i);

    return values.release();
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

RAII (Adquisición de Recursos es Inicialización)

Si algo sale mal ahora durante el cálculo de los valores (es decir, si `DoComputeValue(i)` lanza una excepción), el compilador garantiza que se llame al destructor del objeto RAII, lo que a su vez garantiza que la memoria del objeto array se libera correctamente. Una vez que se han calculado todos los valores, nuevamente entregamos el array double a la invocador como antes, junto con la responsabilidad de eliminarla. Tenga en cuenta que si no hubiéramos llamado a `release()` antes de regresar, el destructor del objeto `DoubleArrayList` aún estaría eliminando el array.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

RAII (Adquisición de Recursos es Inicialización)

Incluso si su programa no funciona con excepciones, se recomienda usar siempre el idioma RAII para administrar sus recursos de manera segura. Las fugas debidas a excepciones pueden ser un punto más difícil, es cierto, pero las fugas de recursos pueden manifestarse con la misma facilidad dentro de funciones con múltiples declaraciones de retorno. Sin RAII, es demasiado fácil olvidarse de liberar todos los recursos antes de cada instrucción de retorno, especialmente, por ejemplo, si alguien que no escribió la función originalmente vuelve al código más tarde para agregar una instrucción de retorno adicional.

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Clases RAll estándar para memoria dinámica

En la sección anterior, creamos la clase `DoubleArrayList` para ayudar a ilustrar cómo funciona el idioma. Además, es importante que sepa cómo implementar una clase `RAll` usted mismo. Sin embargo, por supuesto, existen tipos de biblioteca estándar que realizan el mismo trabajo que `DoubleArrayList`. En la práctica, por lo tanto, nunca escribiría una clase `RAll` para administrar arrays. Un primer tipo de este tipo es `std::unique_ptr<T[]>`. Si incluye el encabezado `<memory>`, puede escribir `ComputeValues()` de la siguiente manera:

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Clases RAII estándar para memoria dinámica

```
double* ComputeValues(size_t howMany)
{
    auto values = std::make_unique<double[]>(howMany); // tipo unique_ptr<double[]>

    for (size_t i = 0; i < howMany; ++i)
        values[i] = DoComputeValue(i);

    return values.release();
}
```

De hecho, con `std::unique_ptr<>`, una opción aún mejor sería escribirlo así:

```
std::unique_ptr<double[]> ComputeValues(size_t howMany)
{
    auto values = std::make_unique<double[]>(howMany);

    for (size_t i = 0; i < howMany; ++i)
        values[i] = DoComputeValue(i);

    return values;
}
```

ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Clases RAII estándar para memoria dinámica

Si devuelve el `unique_ptr<>` desde `ComputeValues()`, por supuesto, tendrá que ajustar ligeramente la función `main()` en consecuencia. Si lo hace, notará que ya no necesita la instrucción `delete[]` allí. ¡Esa es otra fuente potencial de fugas de memoria eliminada! A menos que esté pasando un recurso, por ejemplo, a una función heredada, rara vez hay necesidad de liberar un recurso de su objeto RAII. Simplemente pase el objeto RAII en sí.

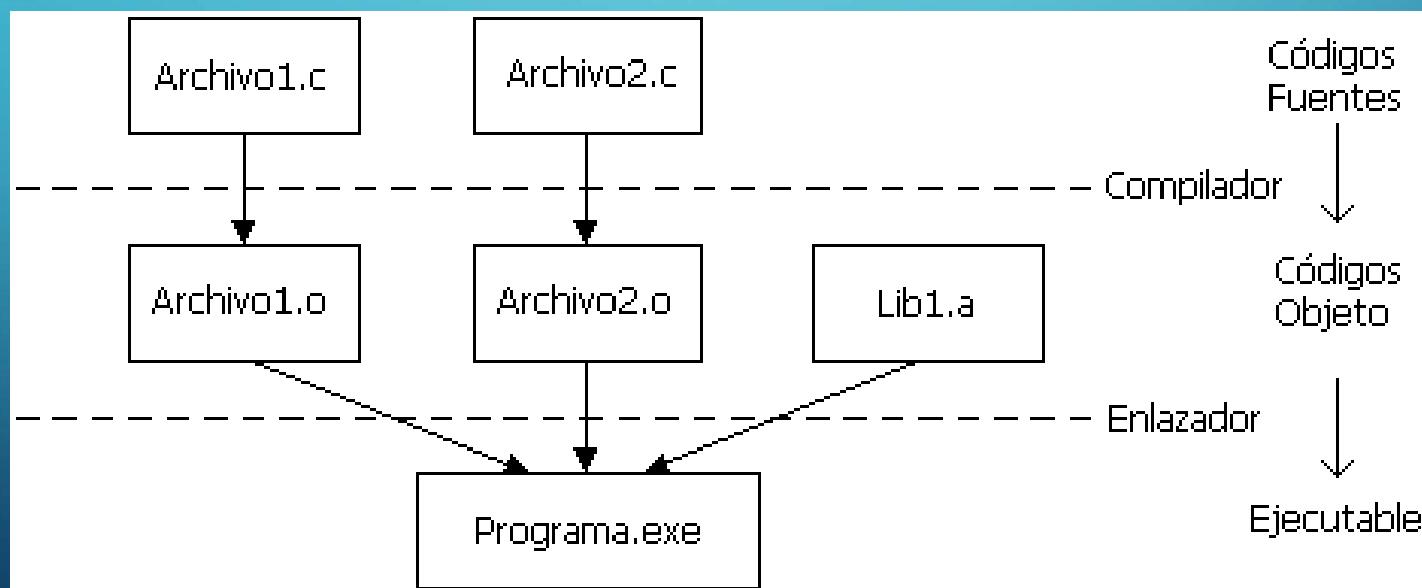
ERRORES EN TIEMPO DE EJECUCION Y EXCEPCIONES

Clases RAII estándar para memoria dinámica

Toda la memoria dinámica debe ser administrada por un objeto RAII. La biblioteca estándar ofrece punteros inteligentes (como `std::unique_ptr<>` y `shared_ptr<>`) y contenedores dinámicos (como `std::vector<>`) para este propósito. En el caso de los punteros inteligentes, siempre se deben usar `make_unique()` y `make_shared()` en lugar de `new` / `new[]`. Por lo tanto, una consecuencia importante de estas pautas es que los operadores `new`, `new[]`, `delete` y `delete[]` generalmente ya no tienen cabida en un programa C++ moderno. Utilice siempre RAII.

PROCESO DE COMPILACION

Los proyectos suelen estar compuestos de uno o mas archivos de código fuente. Compilar cada uno de estos archivos resulta en archivos .o, llamados código objeto. Estos archivos contienen el código fuente traducido a código máquina. Este ejemplo muestra archivos de código fuente escritos en C, pero el proceso para C++ es idéntico.



PROCESO DE COMPILACION

Luego de tener todos los archivos .o resultantes (y los .a que corresponden a bibliotecas estáticas), es necesario unirlos para formar el .exe final, usando un enlazador. Este proceso se encarga de unirlos, haciendo que las referencias entre las distintas partes funcione correctamente.

Si nos referimos g++, el comando ar nos permite crear bibliotecas estáticas.

Ejemplo: Los siguientes comandos compilan el archivo fuente main con niveles de optimización 0 y 2.

```
g++ -o app -Wall main.cpp -O0
```

(`-O1`, `-O2`, `-O3`, `-Os`, `-Ofast`)

```
g++ -o app -Wall main.cpp -O2
```

PROCESO DE COMPILACION

Si se omite la opción `-O`, se utiliza `-O0`, que significa que no hay optimizaciones, como valor predeterminado (la especificación de `-O` sin un número se resuelve en `-O1`).

Compilación y enlazado:

Opción 1:

```
g++ -o nombre_ejecutable first.cpp second.cpp third.cpp
```

Opción 2:

```
g++ -c first.cpp  
g++ -c second.cpp  
g++ -c third.cpp  
g++ -o nombre_ejecutable first.o second.o third.o [otras dependencias]
```

PROCESO DE COMPILACION

Ejemplo: Compilación y enlazado.

```
g++ -std=c++14 code.cpp stats.cpp lib/static_library.a
```

1. Se compilan code.cpp, stats.cpp y se obtienen code.o, stats.o aplicando el estándar C++14.
2. Los códigos objeto code.o, stats.o se enlazan a static_library.a y se obtiene el ejecutable.