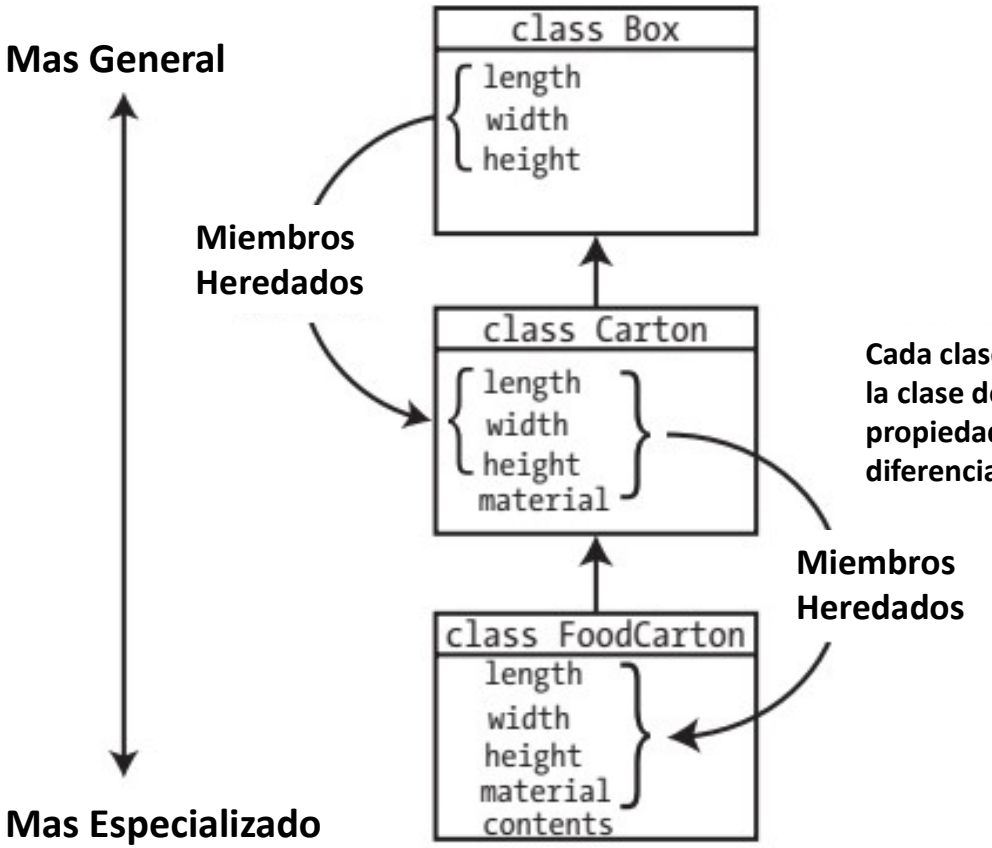
C++ SEMANA 3

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 que tipo de cartógn esta hecho).

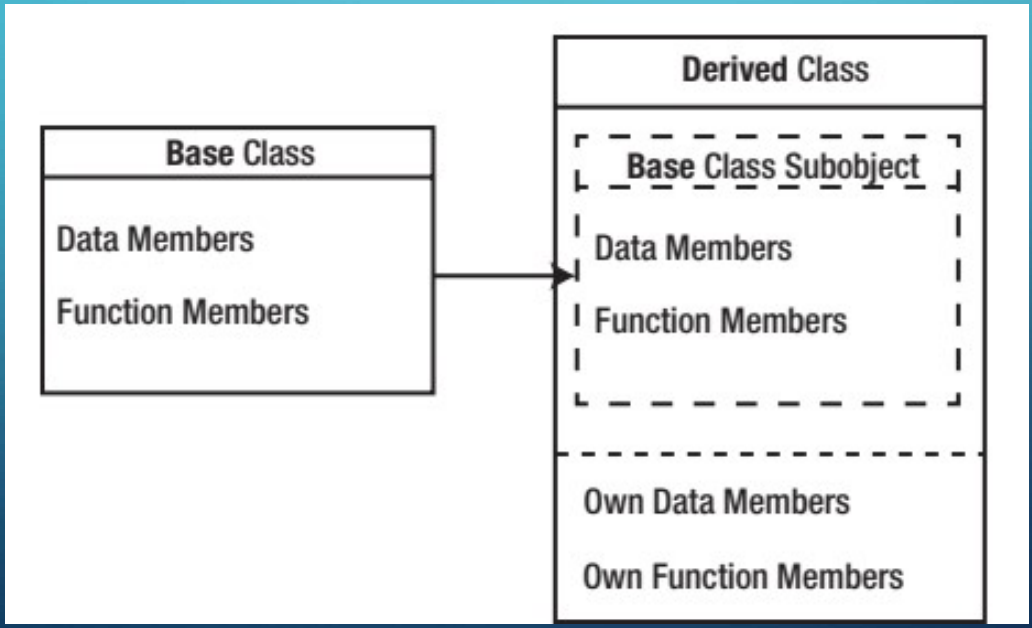
Luego, podría especializarse aún más utilizando la definición Carton para describir una clase FoodCarton, que es un tipo especial de Carton que está diseñado para contener alimentos. Un objeto FoodCarton tendrá todas las propiedades de un objeto Carton y un miembro adicional para modelar los contenidos.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

HERENCIA


Mas General
class Box
length
width
height
Miembros
Heredados
class Carton
length
width
height
material
Cada clase tiene las propiedades de la clase de la que se deriva, más propiedades adicionales que lo diferencian
Miembros
Heredados
class FoodCarton
length
width
height
material
contents
Mas Especializado

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 sobrecara 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_view 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

```cpp
#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 << "candyCarton occupies " << 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;

    // Uncomment any of the following for an error...
    // box.length = 10.0;
    // chocolateCarton.length = 10.0;

    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:

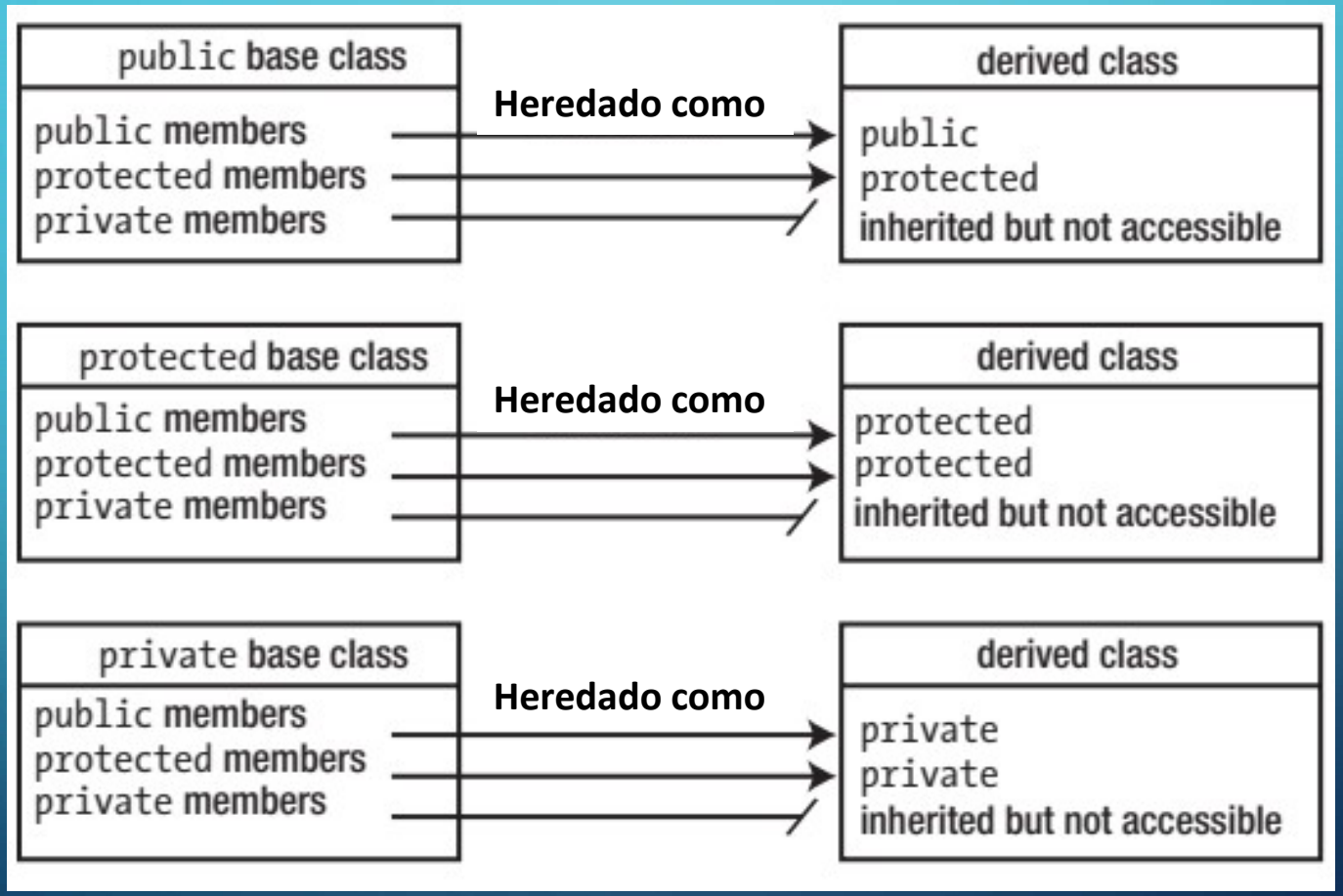
HERENCIA

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


public base class
public members
protected members
private members
Heredado como
derived class
public
protected
inherited but not accessible
protected base class
public members
protected members
private members
Heredado como
derived class
protected
protected
inherited but not accessible
private base class
public members
protected members
private members
Heredado como
derived class
private
private
inherited but not accessible

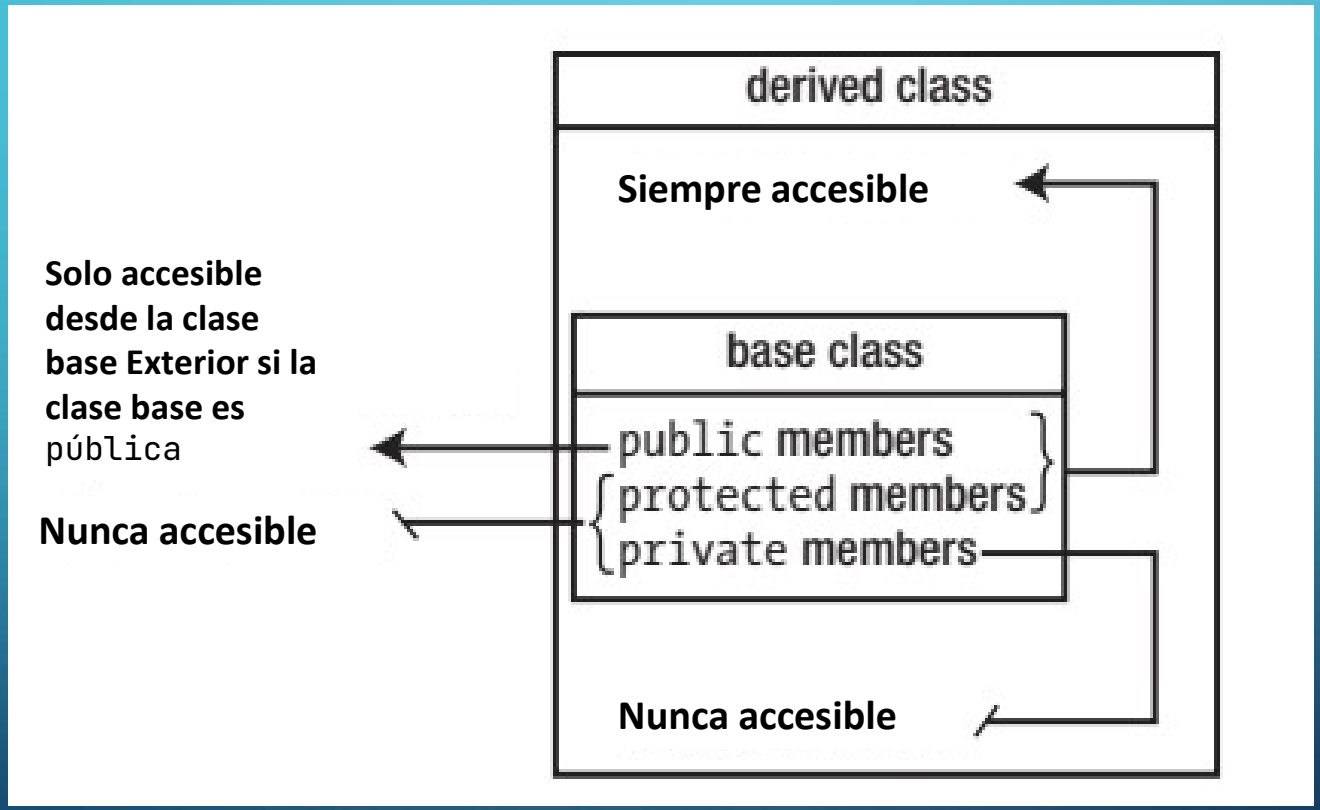
HERENCIA

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

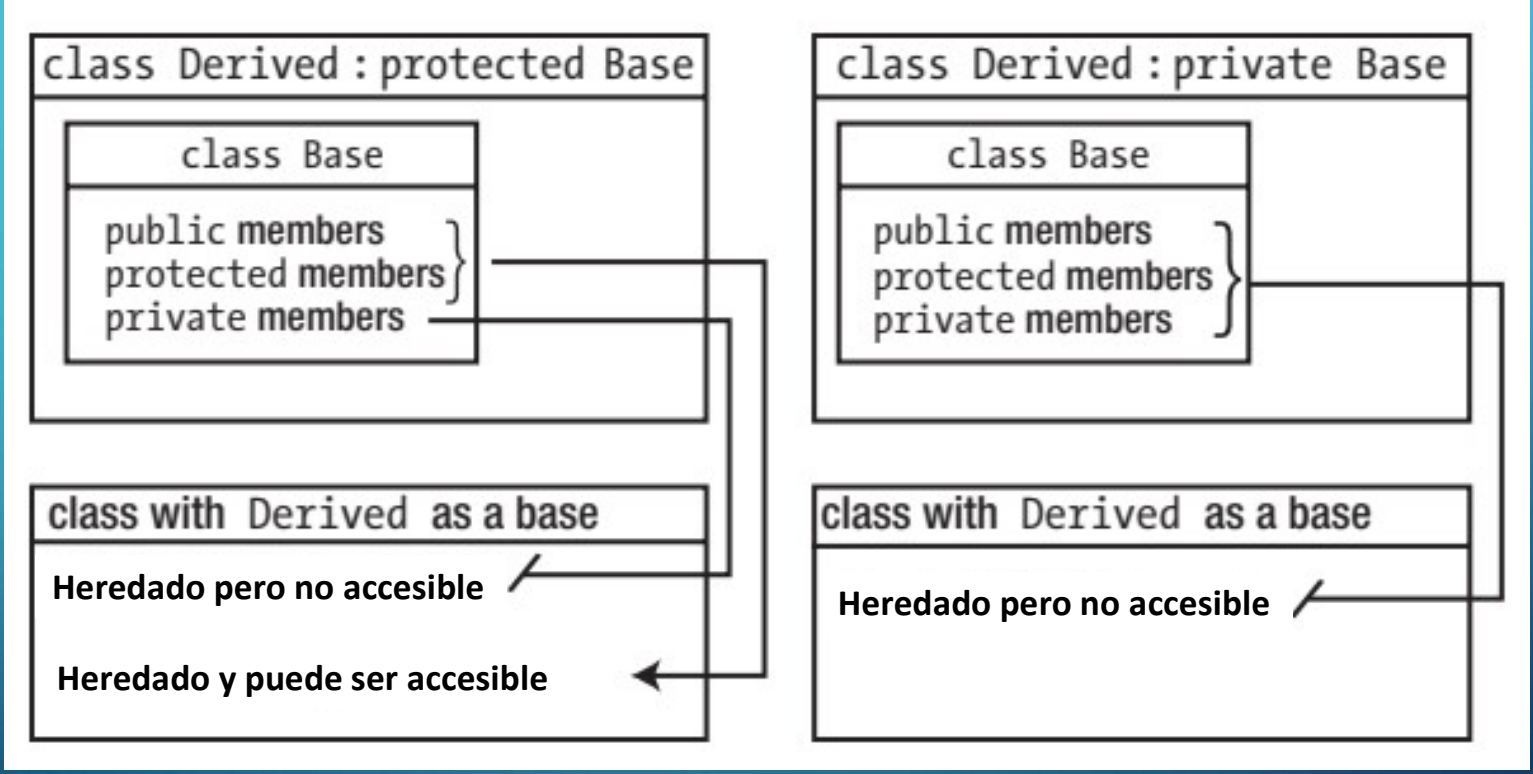
Especificadores de acceso y jerarquías de clases

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 mas 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 protegidas 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 leen.

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_view 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 publico 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.

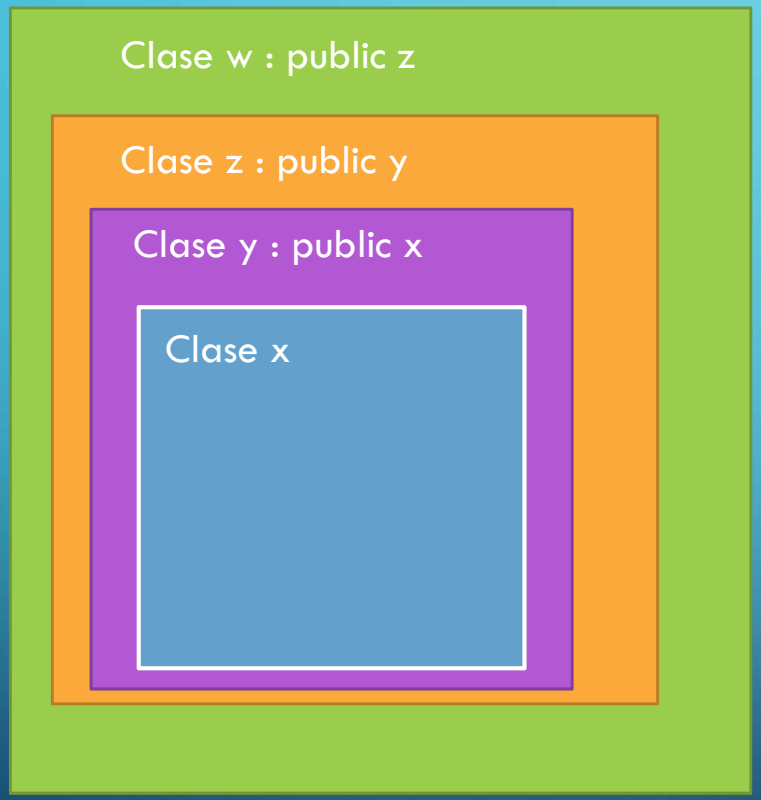
HERENCIA

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 argurmentos). 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 imrpime 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 imrpime 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

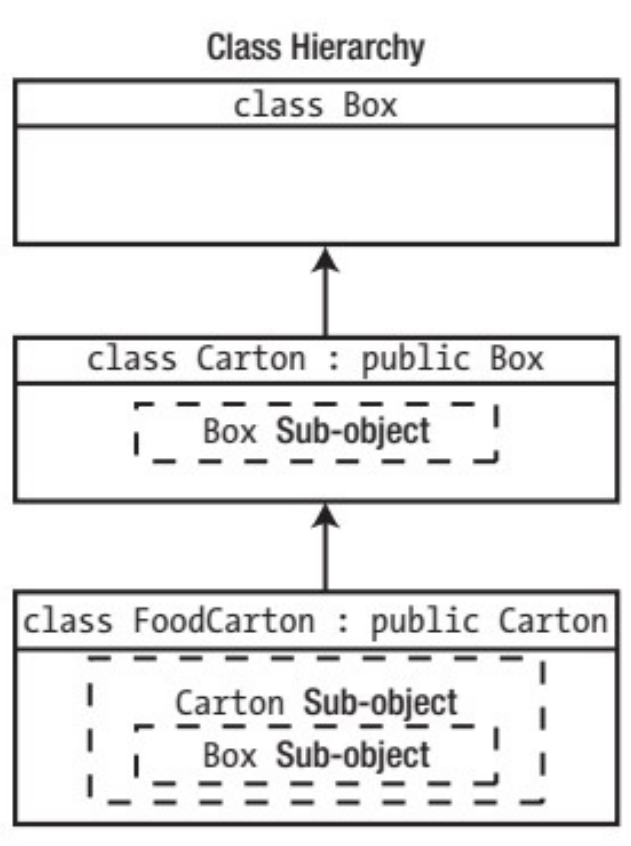
El orden en el cual los destructores son llamados

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 destructores son llamados



Secuencia de constructores para crear un objeto FoodCarton:

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

Secuencia de destructores 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.