Programación en C++/Herencia

De Wikilibros, la colección de libros de texto de contenido libre.
< Programación en C++

Editores:

← Sobrecarga de Operadores Funciones virtuales →

Contenido

Introducción

La 'herencia' es una de las piedras angulares de la POO ya que ésta permite la creación de clasificaciones jerárquicas. Con la herencia, es posible crear una clase general que defina tratos comunes a una serie de elementos relacionados. Esta clase podría luego ser heredada por otras clases más específicas, cada una agregando solo aquellas cosas que son únicas para la clase 'heredera'.

En terminología estándar C++, una clase que es heredada es referida como la clase 'base'. La clase que efectúa la herencia es llamada la clase 'derivada'. Además, una clase derivada puede ser usada como una clase base por otra clase derivada. De esta manera, una jerarquía multicapa de clases puede ser lograda.

INTRODUCIENDO LA HERENCIA.

C++ soporta herencia permitiendo a una clase incorporar otra clase dentro de su declaración. Antes de discutir los detalles y la teoría, se procede a comenzar por un ejemplo de herencia. La siguiente clase, llamada 'VehiculoRodante', define muy ampliamente a vehículos que viajan por la carretera. Este almacena el numero de ruedas que un vehículo tiene y el numero de pasajeros que puede llevar.

Un regla sencilla para recordar esto es: "Una clase derivada hereda de una clase base"

```
// Definición de una clase base para vehiculos
class VehiculoRodante
{
public:
 // CICLO DE VIDA
 /* En este lugar se sitúan los constructores, los destructores, y/o los constructor
es copia */

 // OPERADORES
 /* Aquí van los métodos que se apliquen sobre operadores */

 // OPERACIONES
 /* Aquí van los métodos de esta clase que no sean ni de acceso ni de petición o tra
tamiento */

 // ACCESO
 /* Aquí van las funciones de acceso a los datos miembro o variables propias del obj
eto */

 /*
 * Función 'set_ruedas'
 * Recibe: num como int
 * Devuelve: void
 * Asigna al dato miembro 'mRuedas' el valor 'num'
 */
 void set_ruedas(int num)
```

```
{
 this->mRuedas = num;
}

/*
* Función 'get_ruedas'
* Recibe: void
* Devuelve: int
* Devuelve el valor que hay dentro del dato miembro 'mRuedas'
*/
int get_ruedas(void)
{
 return this->mRuedas;
}

/*
* Función 'set_pasajeros'
* Recibe: num como int
* Devuelve: void
* Asigna al dato miembro 'mPasajeros' el valor 'num'
*/
void set_pasajeros(int num)
{
 this->mPasajeros = num;
}

/*
* Función 'get_pasajeros'
* Recibe: void
* Devuelve: int
* Devuelve el valor que hay dentro del dato miembro 'mPasajeros'
*/
int get_pasajeros(void)
{
 return this->mPasajeros;
}

// PETICIONES/TRATAMIENTOS
/* Aquí van las funciones del tipo "Is", que generalmente devuelven true/false */

private:
 /* Generalmente en 'private' se sitúan los datos miembros */
 int mRuedas;
 int mPasajeros;
};
```

Se puede usar esta amplia definición de un vehículo rodante para ayudar a definir tipos específicos de vehículos. Por ejemplo, el fragmento de código mostrado aqui podría usarse para ser heredado por una clase derivada llamada 'Camion'.

```
// Definición de una clase 'Camion' derivada de la clase base 'VehiculoRodante'.
class Camion : public VehiculoRodante
{

public:
 // CICLO DE VIDA
 /* En este lugar se sitúan los constructores, los destructores, y/o los constructor
es copia */

 // OPERADORES
 /* Aquí van los métodos que se apliquen sobre operadores */
```

```
 // OPERACIONES
 /* Aquí van los métodos de esta clase que no sean ni de acceso ni de petición o tra
tamiento */

 // ACCESO
 /* Aquí van las funciones de acceso a los datos miembro o variables propias del obj
eto */

 /*
 * Función 'set_carga'
 * Recibe: size como int
 * Devuelve: void
 * Asigna al dato miembro 'mCarga' el valor 'size'
 */
 void set_carga(int size)
 {
  this->mCarga = size;
 }

 /*
 * Función 'get_carga'
 * Recibe: void
 * Devuelve: int
 * Devuelve el valor que hay dentro del dato miembro 'mCarga'
 */
 int get_carga(void)
 {
  return this->mCarga;
 }


 /*
 * Función 'Mostrar'
 * Recibe: void
 * Devuelve: void
 * Muestra por pantalla las ruedas, pasajeros y la capacidad de carga del objeto 'Ca
mion'
 */
 void Mostrar(void);

 // PETICIONES/TRATAMIENTOS
 /* Aquí van las funciones del tipo "Is", que generalmente devuelven true/false */

private:
 /* Generalmente en 'private' se sitúan los datos miembros */
 int mCarga;
};
```

Como 'Camion' hereda de 'VehiculoRodante', 'Camion' incluye todo de 'vehiculo_rodante'. Entonces agrega 'carga' a la misma, en conjunto con las funciones miembros que trabajen sobre este dato.

Nótese como 'VehiculoRodante' es heredado. La forma general para la herencia se muestra aquí:

```
class ClaseDerivada : acceso ClaseBase
{
    //cuerpo de la nueva clase
}
```

Aquí, 'acceso' es opcional. Sin embargo, si se encuentra presente, debe ser 'public', 'private', o 'protected'. Se aprenderá más sobre estas opciones más tarde, por ahora para todas las clases heredadas se usará el acceso 'public'. Usar public significa que todos los miembros públicos de la clase base serán también miembros públicos de la clase derivada. Por tanto, en el ejemplo anterior, miembros de la clase 'Camion' tienen acceso a los miembros públicos de 'VehiculoRodante', justo como si ellos hubieran sido declarados dentro de 'Camion'. Sin embargo, 'camion' no tiene acceso a los miembros privados de 'VehiculoRodante'. Por ejemplo, 'Camion' no tiene acceso a ruedas.

He aquí un programa que usa herencia para crear dos subclases de 'VehiculoRodante'. Una es 'Camion' y la otra es 'Automovil'.

```
// Programa que demuestra la herencia.

// INCLUDES DE SISTEMA
//
#include <iostream>

// INCLUDES DEL PROYECTO
//

// INCLUDES LOCALES
//

// DECLARACIONES
//

// Definición de una clase base para vehiculos
class VehiculoRodante
{
public:
 // CICLO DE VIDA
 /* En este lugar se sitúan los constructores, los destructores, y/o los constructor
es copia */

 // OPERADORES
 /* Aquí van los métodos que se apliquen sobre operadores */

 // OPERACIONES
 /* Aquí van los métodos de esta clase que no sean ni de acceso ni de petición o tra
tamiento */

 /*
 * Función 'set_ruedas'
 * Recibe: num como int
 * Devuelve: void
 * Asigna al dato miembro 'mRuedas' el valor 'num'
 */
 void set_ruedas(int num)
 {
  this->mRuedas = num;
 }

 /*
 * Función 'get_ruedas'
 * Recibe: void
 * Devuelve: int
 * Devuelve el valor que hay dentro del dato miembro 'mRuedas'
 */
```

```
 int get_ruedas(void)
 {
  return this->mRuedas;
 }

 /*
 * Función 'set_pasajeros'
 * Recibe: num como int
 * Devuelve: void
 * Asigna al dato miembro 'mPasajeros' el valor 'num'
 */
 void set_pasajeros(int num)
 {
  this->mPasajeros = num;
 }

 /*
 * Función 'get_pasajeros'
 * Recibe: void
 * Devuelve: int
 * Devuelve el valor que hay dentro del dato miembro 'mPasajeros'
 */
 int get_pasajeros(void)
 {
  return this->mPasajeros;
 }

 // PETICIONES/TRATAMIENTOS
 /* Aquí van las funciones del tipo "Is", que generalmente devuelven true/false */

private:
 /* Generalmente en 'private' se sitúan los datos miembros */
 int mRuedas;
 int mPasajeros;
};

// Definición de una clase 'Camion' derivada de la clase base 'VehiculoRodante'.
class Camion : public VehiculoRodante
{

public:
 // CICLO DE VIDA
 /* En este lugar se sitúan los constructores, los destructores, y/o los constructor
es copia */

 // OPERADORES
 /* Aquí van los métodos que se apliquen sobre operadores */

 // OPERACIONES
 /* Aquí van los métodos de esta clase que no sean ni de acceso ni de petición o tra
tamiento */

 // ACCESO
 /* Aquí van las funciones de acceso a los datos miembro o variables propias del obj
eto */

 /*
 * Función 'set_carga'
 * Recibe: size como int
 * Devuelve: void
 * Asigna al dato miembro 'mCarga' el valor 'size'
 */
 void set_carga(int size)
 {
  this->mCarga = size;
```

```
 }

 /*
 * Función 'get_carga'
 * Recibe: void
 * Devuelve: int
 * Devuelve el valor que hay dentro del dato miembro 'mCarga'
 */
 int get_carga(void)
 {
  return this->mCarga;
 }


 /*
 * Función 'Mostrar'
 * Recibe: void
 * Devuelve: void
 * Muestra por pantalla las ruedas, pasajeros y la capacidad de carga del objeto 'Ca
mion'
 */
 void Mostrar(void);

 // PETICIONES/TRATAMIENTOS
 /* Aquí van las funciones del tipo "Is", que generalmente devuelven true/false */

private:
 /* Generalmente en 'private' se sitúan los datos miembros */
 int mCarga;
};

void Camion::Mostrar(void)
{
 std::cout << "ruedas: " << this->get_ruedas() << std::endl;
 std::cout << "pasajeros: " << this->get_pasajeros() << std::endl;
 std::cout << "Capacidad de carga en pies cúbicos: " << this->get_carga() << std::en
dl;
}

/*
* Este enumerador sirve para definir diferentes tipos de automóvil
*/
enum tipo {deportivo, berlina, turismo};

// Definición de una clase 'Automovil' derivada de la clase base 'VehiculoRodante'.
class Automovil : public VehiculoRodante
{
public:
 // CICLO DE VIDA
 /* En este lugar se sitúan los constructores, los destructores, y/o los constructor
es copia */

 // OPERADORES
 /* Aquí van los métodos que se apliquen sobre operadores */

 // OPERACIONES
 /* Aquí van los métodos de esta clase que no sean ni de acceso ni de petición o tra
tamiento */

 // ACCESO
 /* Aquí van las funciones de acceso a los datos miembro o variables propias del obj
eto */

 /*
 * Función 'set_tipo'
```

```
 * Recibe: t como tipo
 * Devuelve: void
 * Asigna al dato miembro 'mTipoDeAutomovil' el valor 't'
 */
 void set_tipo(tipo t)
 {
  this->mTipoDeAutomovil = t;
 }

 /*
 * Función 'get_tipo'
 * Recibe: void
 * Devuelve: tipo
 * Devuelve el valor que hay dentro del dato miembro 'mTipoDeAutomovil'
 */
 enum tipo get_tipo(void)
 {
  return this->mTipoDeAutomovil;
 };

 /*
 * Función 'Mostrar'
 * Recibe: void
 * Devuelve: void
 * Muestra por pantalla las ruedas, pasajeros y la capacidad de carga del objeto 'Ca
mion'
 */
 void Mostrar(void);

private:
 enum tipo mTipoDeAutomovil;
};

void Automovil::Mostrar(void)
{
 std::cout << "ruedas: " << this->get_ruedas() << std::endl;
 std::cout << "pasajeros: " << this->get_pasajeros() << std::endl;
 std::cout << "tipo: ";

 switch(this->get_tipo())
 {
   case deportivo:
    std::cout << "deportivo";
    break;

   case berlina:
    std::cout << "berlina";
    break;

   case turismo:
    std::cout << "turismo";
  }
 std::cout << std::endl;
}

/*
* Función 'main'
* Recibe: void
* Devuelve: int
* El código es una posible implementación para clarificar el uso efectivo de clases
bases y clases derivadas.
* Muestra por pantalla valores de la clase base y de las clases derivadas.
*/
int main(void)
{
```

```
  Camion Camion1;
  Camion Camion2;
  Automovil Automovil1;

  Camion1.set_ruedas(18);
  Camion1.set_pasajeros(2);
  Camion1.set_carga(3200);

  Camion2.set_ruedas(6);
  Camion2.set_pasajeros(3);
  Camion2.set_carga(1200);

  Camion1.Mostrar();
  std::cout << std::endl;
  Camion2.Mostrar();
  std::cout << std::endl;

  Automovil1.set_ruedas(4);
  Automovil1.set_pasajeros(6);
  Automovil1.set_tipo(tipo::deportivo);

  Automovil1.Mostrar();

  return 0;
}
```

La salida de este programa se muestra a continuación:

ruedas: 18
pasajeros: 2
Capacidad de carga en pies cúbicos: 3200

ruedas: 6
pasajeros: 3
Capacidad de carga en pies cúbicos: 1200

ruedas: 4
pasajeros: 6
tipo: deportivo

Como este programa muestra, la mayor ventaja de la herencia es que permite crear una clase base que puede ser incorporada en clases más específicas. De esta manera, cada clase derivada puede ser precisamente ajustada a las propias necesidades y aun siendo parte de la clasificación general.

Por otra parte, nótese que ambos, 'Camion' y 'Automovil', incluyen una función miembro llamada 'Mostrar()', la cual muestra información sobre cada objeto. Esto ilustra un aspecto del polimorfismo. Como cada función 'Mostrar()' esta enlazada con su propia clase, el compilador puede fácilmente indicar cuál llamar para cualquier objeto dado.

Ahora que se ha visto los procedimientos básicos por los cuáles una clase hereda de otra, se examinará la herencia en detalle.

CONTROL DE ACCESO DE LA CLASE BASE.

Cuando una clase hereda de otra, los miembros de la clase base se convierten en miembros de la clase derivada. El estado de acceso de los miembros de la clase base dentro de la clase derivada es determinado por el especificador de acceso usado para heredar la clase base. El especificador de acceso de la clase base debe ser 'public', 'private' o 'protected'. Si el especificador de acceso no es usado, entonces se usara private por defecto si la clase derivada es una clase. Si la clase derivada es una 'struct' entonces 'public' es el acceso por defecto por la ausencia de un expecificador de acceso explícito. Examinemos las ramificaciones de usar accesos public o private. (El especificador 'protected' se describe en la próxima sección.)

Cuando una clase base es heredada como 'public', todos los miembros públicos de la clase base se convierten en miembros de la clase derivada. En todos los casos, los elementos privados de la clase base se mantienen de esa forma para esta clase, y no son accesibles por miembros de la clase derivada. Por ejemplo, en el siguiente programa, los miembros públicos de 'base' se convierten en miembros publics de 'derivada'. Encima, son accesibles por otras partes del programa.

```
#include <iostream>
using namespace std;

class base {

 int i, j;

public:
 void set(int a, int b) { i = a; j = b; }
 void mostrar() { cout << i << " " << j << "\n"; }
};

class derivada : public base {

 int k;

public:
 derivada(int x) { k = x; }
 void mostrar_k() { cout << k << "\n"; }
};

int main()
{
 derivada obj(3);

 obj.set(1, 2); // accesar a miembro de base
 obj.mostrar(); // accesar a miembro de base

 obj.mostrar_k(); // usa miembro de la clase derivada

return 0;
```

```
}
```

Como set() y mostrar() son heredadas como 'public', ellas pueden ser llamadas en un objeto del tipo 'derivada' desde main(). Como i y j son especificadas como 'private', ellas se mantienen privadas a base.

El opuesto de herencia publica es herencia privada. Cuando la clase base es heredad como privada, entonces todos los miembros públicos de la clase base se convierten en miembros privados de la clase derivada. Por ejemplo, el programa mostrado a continuación no compilara, porque set() y mostrar() son ahora miembros privados de 'derivada', y por ende no pueden ser llamados desde main().

```
TIP:"Cuando una clase base es heredada como 'private' sus miembros públicos se
     convierten en miembros privados de la clase derivada."
```

```
// Este programa no compilara.
#include <iostream>

using namespace std;

class base {

 int i, j;

public:
 void set(int a, int b) { i = a; j = b; }
 void mostrar() { cout << i << " " << j << "\n"; }
};

// Miembros publicos de 'base' son privados en 'derivada'
class derivada : private base {

 int k;

public:

 derivada(int x) { k = x; }
 void mostrar_k() { cout << k << "\n"; }
};

int main()
{

 derivada obj(3);

 obj.set(1, 2);   // Error, no se puede acceder a set()
 obj.mostrar();      // Error, no se puede acceder a mostrar()

return 0;
}
```

La clave a recordar es que cuando una clase base es heredada como 'private', los miembros públicos de la clase base se convierten en miebros privados de la clase derivada. Esto significa que aun ellos

son accesibles por miembros de la clase derivada, pero no pueden ser accedidos por otras partes de su programa.

USANDO MIEMBROS PROTEGIDOS.

En adición a public y private, un miembro de la clase puede ser declarado como protegido. Además, una clase base puede ser heredada como protegida. Ambas de estas acciones son cumplidas por el especificador de acceso 'protected'. La palabra clave 'protected' esta incluida en C++ para proveer gran flexibilidad para el mecanismo de herencia.

Cuando un miembro de una clase es declarado como 'protected', ese miembro, no es accesible a otros elementos no-miembros de la clase en el programa. Con una importante excepción, el acceso a un miembro protegido es lo mismo que el acceso a un miembro privado, este puede ser accedido solo por otros miembros de la clase de la cual es parte. La única excepción a esta regla es cuando un miembro protegido es heredado. En este caso, un miembro protegido difiere sustancialmente de uno privado.

Como debe conocer, un miembro privado de una clase base no es accesible por cualquier otra parte de su programa, incluyendo cualquier clase derivada. Sin embargo, los miembros protegidos se comportan diferente. Cuando una clase base es heredada como publica, los miembros protegidos en la clase base se convierten en miembros protegidos de la clase derivada, y 'son' accesibles a la clase derivada. Además, usando 'protected' usted puede crear miebros de clases que son privados para su clase, pero que aun asi pueden ser heredados y accedidos por una clase derivada.

Considere este programa de ejemplo:

```
#include <iostream>

using namespace std;

class base {

protected:
 int i, j;  // privados a base, pero accesibles a derivada.

public:
 void set(int a, int b) { i = a; j = b; }
 void mostrar() { cout << i << " " << j << "\n"; }
};

class derivada : public base {

 int k;

public:
 // derivada puede accesar en base a 'j' e 'i'
 void setk() { k = i * j; }
 void mostrark() { cout << k << "\n"; }
};
```

```
int main()
{
  derivada obj;

  obj.set(2, 3); // OK, conocido por derivada.
  obj.mostrar(); // OK, conocido por derivada.

  obj.setk();
  obj.mostrark();

return 0;
}
```

Aquí, porque 'base' es heredada por 'derivada' como pública, y porque j e i son declaradas como protegidas, la función setk() en 'derivada' puede acceder a ellas. Si j e i hubieran sido declaradas como privadas por 'base', entonces 'derivada' no tuviera acceso a ellas, y el programa no compilaría.

RECUERDE: El especificador 'protected' le permite crear un miembro de la clase que es accesible desde la jerarquía de la clase, pero de otra manera es privado.

Cuando una clase derivada es usada como clase base para otra clase derivada, entonces cualquier miembro protegido de la clase base inicial que es heredado (como public) por la primera clase derivada puede ser heredado nuevamente, como miembro protegido, por una segunda clase derivada. Por ejemplo, el siguiente programa es correcto, y derivada2 tiene, de hecho, acceso a 'j' e 'i':

```
#include <iostream>
using namespace std;

class base {

protected:
 int i, j;

public:

 void set(int a, int b) { i = a; j = b; }
 void mostrar() { cout << i << " " << j << "\n"; }
};

// j e i se heredan como 'protected'
class derivada1 : public base {
 int k;

public:

 void setk() { k = i*j; } // legal
 void mostrark() { cout << k << "\n"; }
};

// j e i se heredan indirectamente a través de derivada1
```

```
class derivada2 : public derivada1 {
int m;

public:
 void setm() { m = i-j; } // legal
 void mostrarm() { cout << m << "\n"; }
};

int main()
{

 derivada1 obj1;
 derivada2 obj2;

 obj1.set(2, 3);
 obj1.mostrar();
 obj1.setk();
 obj1.mostrark();

 obj2.set(3, 4);
 obj2.mostrar();
 obj2.setk();
 obj2.setm();
 obj2.mostrark();
 obj2.mostrarm();

return 0;
}
```

Cuando una clase base es heredada como 'private', miembros protegidos de la clase base se convierten en miembros privados de la clase derivada. Además, en el ejemplo anterior, si 'base' fuera heredada como 'private', entonces todos los miembros de 'base' se hubieran vuelto miembros privados de derivada1, significando que ellos no podrían estar accesibles a derivada2. (Sin embargo, j e i podrían aun ser accesibles a derivada1.) Esta situación es ilustrada por el siguiente programa, el cual es un error (y no compilara). Los comentarios describen cada error.

```
// Este programa no compilara.
#include <iostream>
using namespace std;

class base {
protected:
 int i, j;

public:

 void set(int a, int b) { i = a; j = b; }
 void mostrar() { cout << i << " " << j << "\n"; }
};

// Ahora todos los elementos de base son privados en derivada1.
class derivada1 : private base {
 int k;

public:
 // Esto es legal porque j e i son privadas a derivada1
 void setk() { k = i*j; } // OK
 void mostrark() { cout << k << "\n"; }
```

```
};

// Acceso a j, i, set() y mostrar no heredado
class derivada2 : public derivada1 {
 int m;

public:
 // Ilegal porque j e i son privadas a derivada1
 setm() { m = j-i; } // error
 void mostrarm() { cout << m << "\n"; }
};

int main()
{
  derivada1 obj1;
  derivada2 obj2;

  obj1.set(1, 2); // Error, no se puede usar set()
  obj1.mostrar();     // Error, no se puede usar show()

  obj2.set(3, 4);  // Error, no se puede usar set()
  obj2.mostrar(); // Error, no se puede usar show()

return 0;
}
```

Incluso aunque 'base' es heredada como privada por derivada1, derivada2 aun tiene acceso a los elementos publicos y protegidos de 'base'. Sin embargo, no puede sobrepasar este privilegio a todo lo largo. Esta es la razon de las partes 'protected' del lenguaje C++. Este provee un medio de proteger ciertos miembros de ser modificados por funciones no-miembros, pero les permite ser heredadas.

El especificador 'protected' puede ser usado con estructuras. Sin embargo no puede ser usado con una 'union' porque la union no puede heredar otra clase o ser heredad. (Algunos compiladores aceptaran su uso en una declaracion en una union, pero como las uniones no pueden participar en la herencia, 'protected' es lo mismo que 'private' en este contexto.)

El especificador 'protected' puede ocurrir en cualquier lugar en la declaracion de una clase, aunque tipicamente ocurre despues de (por defecto) que los miembros privados son declarados, y antes de los miebros publicos. Ademas, la mayor forma completa de la declaraciond de una clase es

```
class nombre-clase {

 miembros privados

protected:
 miembros protegidos

public:
 miembros publicos
};
```

Por supuesto, la categoria protected es opcional.

USANDO PROTECTED PARA LA HERENCIA DE UNA CLASE BASE.

Como adición a especificar el estado protegido para los miembros de una clase, la palabra clave 'protected' tambien puede ser usada para heredar una clase base. Cuando una clase base es heredada como protected, todos los miembros publicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada. Aqui hay un ejemplo:

```
// Demuestra la herencia de una clase base protegida
#include <iostream>
using namespace std;

class base {
 int i;

protected:
 int j;

public:
 int k;
 void seti(int a) { i = a; }
 int geti() { return i; }
};

// heredar 'base' como protected.
class derivada : protected base {

public:
 void setj(int a) { j = a; }; // j es protected aqui.
 void setk(int a) { k = a; }; // k es tambien protected.
 int getj() { return j; }
 int getk() { return k; }
};

int main()
{
  derivada obj;

  /* La proxima linea es ilegal porque seti() es
     un miembro protegido de derivada, lo cual lo
     hace inaccesible fuera de derivada. */
  // obj.seti(10);

  // cout << obj.geti(); // ilegal -- geti() es protected.
  // obj.k = 10;  // tambien ilegal porque k es protected.

    // estas declaraciones son correctas
    obj.setk(10);
    cout << obj.getk() << " ";
    obj.setj(12);
    cout << obj.getj() << " ";

return 0;
}
```

Como puede ver leyendo los comentarios en este programa, k,j, seti() y geti() en 'base' se convierten miembros 'protected' en 'derivada'. Esto significa que ellos no pueden ser accesados por codigo fuera de 'derivada'. Ademas, dentro de main(), referencias a estos miembros a traves de obj son ilegales.

REVISANDO public, protected, y private

Ya que los permisos de acceso que son definidos por 'public', 'protected', y 'private' son fundamentales para la programación en C++, volvamos a revisar sus significados.

Cuando una clase miembro es declarada como 'public', esta puede ser accedida por cualquier otra parte del programa. Cuando un miembro es declarado como 'private', este puede ser accedido solo por miembros de su clase. Además, clases derivadas no tienen acceso a miembros privados de la clase base. Cuando un miembro es declarado como 'protected', este puede ser accedido solo por miembros de su clase, o por clases derivadas. Además, protected permite que un miembro sea heredado, pero que se mantenga privado en la jerarquía de la clase.

Cuando una clase base es heredada por el uso de 'public', sus miembros publicos se convierten en miebros publicos de la clase derivada, y sus miembros protected se convierten en miembros protected de la clase derivada.

Cuando una clase base es heredada por el uso de 'protected', sus miembros publicos y protegidos se convierten en miembros protected de la clase derivada.

Cuando una clase base es heredada por el uso de 'private', sus miembros publicos y protegidos se convierten en miebros private de la clase derivada.

En todos los casos, los miembros privados de la clase base se mantienen privados a la clase base, y no son heredados.

HEREDANDO MULTIPLES CLASES BASE

Es posible para una clase derivada heredar dos o mas clases base. Por ejemplo, en este corto programa, derivada hereda de ambas clases base1 y base2:

```
// Un ejemplo de multiples clases base
#include <iostream>
using namespace std;

class base1 {
protected:
 int x;
int m;

public:
 void showx() { cout << x << "\n"; }
};
```

```
class base2 {
protected:
 int y;

public:
 void showy() { cout << y << "\n"; }
};

// Heredar multiples clases base.
class derivada : public base1, public base2 {

public:
 void set(int i, int j) { x= i; y = j; };
};

int main()
{

  derivada obj;

  obj.set(10, 20);  // proveida por derivada.
  obj.showx();      // desde base1
  obj.showy();      // desde base2

return 0;
}
```

Como este ejemplo ilustra, para causar que mas de una clase base sea heredad, debe usarse una lista separada por comas. Ademas, asegurese de usar un especificador de acceso para cada clase heredada.

CONSTRUCTORES, DESTRUCTORES, Y HERENCIA

Existen dos importantes preguntas relativas a los constructores y destructores cuando la herencia se encuentra implicada. Primera, ¿dónde son llamados los constructores y destructores de las clases base y clases derivadas? Segunda, ¿como se pueden pasar los parámetros al constructor de una clase base? Esta sección responde estas preguntas.

Examine este corto programa:

```
#include <iostream>
using namespace std;

class base {

public:
 base() { cout << "Construyendo base\n"; }
 ~base() { cout << "Destruyendo base\n"; }
};


class derivada : public base {
public:
  derivada() { cout << "Construyendo derivada\n"; }
  ~derivada() { cout << "Destruyendo derivada\n"; }
};

int main()
```

```
{
  derivada obj;

  // no hacer nada mas que construir y destruir obj

return 0;
}
```

Como el comentario en main() indica, este programa simplemente construye y destruye un objeto llamado obj, el cual es de la clase 'derivada'. Cuando se ejecuta, este programa muestra:

```
Construyendo base
Construyendo derivada
Destruyendo derivada
Destruyendo base
```

Como puede ver, el constructor de 'base' es ejecutado, seguido por el constructor en 'derivada'. A continuación (ya que obj es inmediatamente destruido en este programa), el destructor en 'derivada' es llamado, seguido por el de 'base'.

El resultado del experimento anterior puede ser generalizado como lo siguiente: Cuando un objeto de una clase derivada es creado, el constructor de la clase base es llamado primero, seguido por el constructor de la clase derivada. Cuando un objeto derivada es destruido, su destructor es llamado primero, seguido por el destructor de la clase base. Viéndolo de otra manera, los constructores son ejecutados en el orden de su derivación. Los destructores son ejecutado en orden inverso de su derivación.

Si lo piensa un poco, tiene sentido que las funciones constructor sean ejecutadas en el orden de su derivación. Porque una clase base no tiene conocimiento de las clases derivadas, cualquier inicialización que necesite realizar es separada de, y posiblemente un prerequisito a, cualquier inicialización realizada por la clase derivada. Además, esta debe ejecutarse primero.

Asimismo, es bastante sensible que los destructores sean ejecutados en orden inverso a su derivación. Ya que la clase base contiene una clase derivada, la destruccion de una clase base implica la destruccion de su clase derivada. Además, el constructor derivado debe ser llamado antes de que el objeto sea completamente destruido.

En el caso de una gran jerarquía de clases (ej: cuando una clase derivada se convierta en la base clase de otra clase derivada), la regla general se aplica: Los constructores son llamados en orden de su derivacion, los destructores son llamados en orden inverso. Por ejemplo, este programa

```
#include <iostream>
using namespace std;

class base {

public:
```

```
 base() { cout << "construyendo base\n"; }
 ~base() { cout << "Destruyendo base\n"; }
};

class derivada1 : public base {

public:
 derivada1() { cout << "Construyendo derivada1\n"; }
 ~derivada1() { cout << "destruyendo derivada1\n"; }
};


class derivada2 : public derivada1 {

public:
 derivada2() { cout << "Construyendo derivada2\n"; }
 ~derivada2() { cout << "Destruyendo derivada2\n"; }
};

int main()
{
  derivada2 obj;

  // construir y destruir obj

return 0;
}
```

Muestra la salida:

```
construyendo base
Construyendo derivada1
Construyendo derivada2
Destruyendo derivada2
destruyendo derivada1
Destruyendo base
```

La misma regla general se aplica en situaciones en que se ven implicadas multiples clases base. Por ejemplo, este programa

```
#include <iostream>
using namespace std;

class base1 {

public:
 base1() { cout << "Construyendo base1\n"; }
 ~base1() { cout << "Desstruyendo base1\n"; }
};

class base2 {

public:
 base2() { cout << "Construyendo base2\n"; }
 ~base2() { cout << "Destruyendo base2\n"; }
};

class derivada : public base1, public base2 {
```

```
public:
  derivada() { cout << "Construyendo derivada\n"; }
  ~derivada() { cout << "Destruyendo derivada\n"; }
};

int main()
{
  derivada obj;

  // construir y destruir obj

return 0;
}
```

Produce la salida:

```
Construyendo base1
Construyendo base2
Construyendo derivada
Destruyendo derivada
Destruyendo base2
Destruyendo base1
```

Como puede ver, los constructores son llamados en el orden de su derivacion, de izquierda a derecha, como se especifico en la lista de herencia en derivada. Los destructores son llamados en orden inverso, de derecha a izquierda. Esto significa que si base2 fuera especificada antes de base1 en la lista de derivada, como se muestra aqui:

```
class derivada: public base2, public base1 {
```

entonces la salida del programa anterior hubiera sido asi:

```
Construyendo base2
Construyendo base1
Construyendo derivada
Destruyendo derivada
Destruyendo base1
Derstruyendo base2
```

PASANDO PARAMETROS A LOS CONSTRUCTORES DE LA CLASE BASE

Hasta ahora, ninguno de los ejemplos anteriores han incluido constructores que requieran argumentos. En casos donde solo el constructor de la clase derivada requiera uno o mas argumentos, simplemente use la sintaxis estandarizada de parametrizacion del constructor. Pero, como se le pasan argumentos a un constructor en una clase base? La respuesta es usar una forma expandida de la declaracion de la clase derivada, la cual pasa argumentos entre uno o mas constructores de la clase base. La forma general de esta declaracion expandida se muestra aqui:

```
constructor-derivada(lista-argumentos) : base1(lista-argumentos),
                                         base2(lista-argumentos),
                                         baseN(lista-argumentos)
{

  cuerpo del constructor derivado
}
```

Aqui, desde base1 hasta baseN son los nombres de las clases base heredadas por la clase derivada. Notese los dos puntos separando la declaracion del constructor de la clase derivada de las clases base, y que las clases base estan separadas cada una de la otra por comas, en el caso de multiples clases base.

Considere este programa de ejemplo:

```
#include <iostream>
using namespace std;

class base {

protected:
 int i;

public:
 base(int x) { i = x; cout << "Construyendo base\n"; }
 ~base() { cout << "Destruyendo base\n"; }
};


class derivada : public base {
int j;

public:
  // derivada usa x, y es pasado en consjunto a base
  derivada(int x, int y) : base(y)
     { j = x; cout << "Construyendo derivada\n"; }

  ~derivada() { cout << "Destruyendo derivada\n"; }
  void mostrar() { cout << i << " " << j << "\n"; }
};

int main()
{
  derivada obj(3, 4);

  obj.mostrar();   // muestra 4  3

return 0;
}
```

Aqui, el constructor en derivada es declarado para que tome 2 parametros, x, y. Sin embargo, derivada() usa solo x, y es pasada a base(). En general, el constructor de la clase derivada debe

declarar el/los parametro(s) que esta clase requiere, tamnbien cualquiera requerido por la clase base. Como el ejemplo anterior ilustra, cualquiera de los parametrosr requeridos por la clase base son pasados a esta en la lista de argumentos de la clase base, especificadas despues de los dos puntos.

Aqui tenemos un programa de ejemplo que usa multiples clases base:

```
#include <iostream>
using namespace std;

class base1 {
protected:
 int i;

public:
 base1(int x) { i = x; cout << "Construyendo base1\n"; }
 ~base1() { cout << "Destruyendo base1\n"; }
};

class base2 {
protected:
 int k;

public:
 base2(int x) { k = x; cout << "construyendo base2\n"; }
 ~base2() { cout << "Destuyendo base2\n"; }
};

class derivada : public base1, public base2 {

 int j;

public:
 derivada(int x, int y, int z) : base1(y), base2(z)
    { j = x; cout << "construyendo derivada\n"; }

 ~derivada() {  cout << "Destruyendo derivada\n"; }
 void mostrar() { cout << i << " " << j << " " << k << "\n"; }
};

int main()
{
  derivada obj(3, 4, 5);

  obj.mostrar();  // mostrar 3  4  5

return 0;
}
```

Es importante comprender que los argumentos al constructor de la clase base son pasado via argumentos al constructor de la clase derivada. Ademas, incluso si el constructor de una clase derivada no usa ningun argumento, este aun debe declarar uno o mas argumentos si la clase base toma uno o mas argumentos. En esta situacion, los argumentos pasado a la clase derivada son simplemente pasados hacia la clase base. Por ejemplo en el siguiente programa, el constructor en derivada no toma argumentos, pero base1() y base2() lo hacen:

```
#include <iostream>
```

```
using namespace std;

class base1 {

protected:

 int i;

public:
 base1(int x) { i=x; cout << "Construyendo base1\n"; }
 ~base1() { cout << "Destruyendo base1\n"; }
};

class base2 {
protected:
 int k;

public:
 base2(int x) { k=x; cout << "construyendo base2\n"; }
 ~base2() { cout << "Destruyendo base2\n"; }
};

class derivada : public base1, public base2 {

public:

 /* el constructor en derivada no usa parametros,
    pero aun asi debe ser declarado para tomarlos
    y pasarselos a las clases base.
  */
 derivada(int x, int y) : base1(x), base2(y)
  { cout << "Construyendo derivada\n"; }

 ~derivada() { cout << "Destruyendo derivada\n"; }
 void mostrar() { cout << i << "" << k << "\n"; }
};

int main()
{
  derivada obj(3, 4);

  obj.mostrar();   // mostrar 3  4

return 0;
}
```

El constructor de la clasde derivada es libre de usar cualquiera de todos los parameteos que se declara que toma, ya sea que uno o mas sean pasados a una clase base. Viendolo diferente, solo porque un argumento es pasado a la clase base no declara su uso en la clase derivada. Por ejemplo, este fragmento es perfectamente valido:

```
class derivada : public base {
int j;

public:
 // derivada usa x, y, y tambien las pasa a base
 derivada(int x, int y) : base(x,y)
  { j = x*y; cout << "construyendo derivada\n"; }
// ...
```

Un punto final a tener en mente cuando se pase argumentos a los constructores de la clase base: Un argumento siendo pasado puede consistir de cualquier expresion valida en ese momento, incluyendo llamadas a funciones y variables. Esto es para sostener el hecho de que C++ permite la inicializacion dinamica.

GARANTIZANDO ACCESO

Cuando una clase base es heredada como private, todos los miembros de esa clase (public, protected o private) se convierten en miembros privates de la clase derivada. Sin embargo, en ciertas circunstancias, podra restaurar uno o mas miembros heredados a su especificacion de acceso original. Por ejemplo, quizas desee permitir a ciertos miembros publicos de la clase base tener un estado public en la clase derivada, incluso aunque la clase es heredada como private. Hay dos maneras de lograr esto. Primero, usted deberia usar una declaracion 'using' con la clase derivada. Este es el metodo recomendado por los estandares C++ para uso no nuevo codigo. Sin embargo, una discusion de 'using' se retrasa hasta finales de este libro donde los 'namespaces' son examinados. (La primera razon para usar using es para dar soporte a namespaces.) La segunda manera para ajustar el acceso a un miembro heredado es emplear una declaracion de acceso. Las declaraciones de acceso son aun soportadas por el estandar C++, pero han sido marcadas como deprecadas, lo que significa que ellas no deben ser usadas para nuevo codigo, Como aun son usadas en codigo existente, una discusion de las declaraciones de acceso es presentada aqui:

Una declaracion de acceso toma esta forma general:

```
clase-base::miembro;
```

La declaracion de acceso es ubicada bajo los apropiados encabezados de acceso en la clase derivada. Notese que no se requiere (o permite) una declaracion de tipo en una declaracion de acceso.

para ver como una declaracion de acceso funciona, comenzemos con este corto fragmento:

```
class base {
public:
 int j;  // public en base
};

// heredar base como private
class derivada : private base {
public:

 // aqui esta la declaracion de acceso
 base::j;  // hacer j publica nuevamente.
 // ...
};
```

Como base es heredada como private por derivada, la variable publica j es hecha una variable 'private' en derivada. Sin embargo, la inclusion de esta declaracion de acceso

```
base::j;
```

Puede usar una declaracion de acceso para restaturar los derechos de acceso de miembros 'public' y 'protected', Sin embargo, no puede usar una declaracion de acceso para elevar o disminuir un estado de acceso de un miembro. Por ejemplo, un miembro declarado como private dentro de una clase base no puede ser hecho public por una clase derivada. (Permitir esto destruirira la encapsulacion!)

El siguiente programa ilustra el uso en declaraciones de acceso.

```
#include <iostream>
using namespace std;

class base {
 int i;   // private en base

public:
 int j, k;
 void seti(int x) { i = x; }
 int geti() { return i; }
};

// Heredar base como private
class derivada : private base {

public:
 /* las siguientes tres declaraciones
    sobreescriben la herencia de base
    como private y restablece j, seti()
    y geti() a publico acceso
  */
  base::j;     // hacer j public nuevamente - pero no k
  base::seti;  // hacer seti() public
  base::geti;  // hacer geti() public

 // base::i;   // ilegal - no se puede elevar el acceso

  int a; // public
};

int main()
{

  derivada obj;

  // obj.i = 10;   // ilegal porque i es private en derivada

  obj.j = 20;      // legal porque j es hecho public en derivada
  //obj.k = 30;    // ilegal porque k es private en derivada

  obj.a = 40;      // legal porque a es public en derivada
  obj.seti(10);

  cout << obj.geti() << "  " << obj.j << " " << obj.a;
```

```
return 0;
}
```

Notese como este programa usa declaraciones de acceso para restaurar j, seti(), y geti() a un estadi 'public'. Los comentarios describen otras varias restricciones de acceso.

C++ provee la habilidad de ajustar acceso a los miembros heredados para acomodar aquellas situaciones especiales en las cuales la mayoria de una clase heredad esta pensada para ser private, pero unos cuantos miembros se hacen para mantener su estado public o protected. Es mejor usar esta caracteristica escasamente.

LEYENDO GRAFICOS DE HERENCIA EN c++

En ocasiones las jerarquias de clases en C++ son hechas graficamente para hacerlas mas facil de comprender. Sin embargo, debido a la forma en que estas son usualmente dibujadas por programadores de C++, los graficos de herencia son a veces incomprendidos por principiantes. Por ejemplo, considere una situacion en la cual la clase A es heredad por la clase B, la cual es heredada por la clase C. Usando la notacion estandar C++, esta situacion es dibujada como se muestra aqui:

```
NOTA: "el simbolo ^ es la punta de la fecha,
       que indica direccion."
```

```
                        A
                        ^
                        |
                        B
                        ^
                        |
                        C
```

Como puede ver, las flechas apuntan hacia arriba, no hacia abajo. Mientras que muchas personas encuentran inicialmente la direccion de las flechas contra-intuitivasm este es el estilo que la mayoria de los programadores de C++ han adoptado. En graficos al estilo C++, las flechas apuntan a la clase base. Ademas, la flecha quiere decir "derivada desde", y no "derivando". Aqui tenemos otro ejemplo. Puede describir en palabras que significa esto?

```
                A       B
                ^       ^
                |       |
                ^       ^
                C       D
                ^       ^
                 \     /
                  \   /
                   \ /
                    E
```

Este grafico declara que la clase E es derivada de C y D. (Eso es, E tiene multiples clases base, llamadas C y D.) Ademas, C es derivada de A, y D derivada de B.

Mientras que la direccion de las flechas puede confundir a la primera, es mejor que se sienta familiar con este estilo de notacion grafica, ya que es comunmente usada en libros, revistas, y documentacion de compiladores.

CLASES BASE VIRTUALES

Un elemento de ambiguedad puede ser introducido en un programa C++ cuando multiples clases bases son heredadas. Considere este incorrecto programa:

```
// Este programa contiene un error y no compilara.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derivada 1 hereda base
class derivada1 : public base {

public:
 int j;
};

// derivada2 hereda base
class derivada2 : public base {

public:
 int k;
};

/* derivada3 hereda de ambas, derivada1 y derivada2
   esto significa que hay 2 copias de base en derivada3!
 */

class derivada3 : public derivada1, public derivada2 {
public:

 int sum;
};

int main()
{

  derivada3 obj;

  obj.i = 10;    // esto es ambiguo; cual i???
  obj.j = 20;
  obj.k = 30;

  // i es ambiguo aqui, tambien
  obj.sum = obj.i + obj.j + obj.k;

  // tambien ambiguo, cual i?
  cout << obj.i << " ";
```

```
  cout << obj.j << " " << obj.k << " ";
  cout << obj.sum;

return 0;
}
```

Como indican los comentarios en este programa, ambos derivada1 y derivada2 heredan base. Sin embargo, derivada3 hereda ambos, derivada1 y derivada2. Como resultado existen dos copias de base presenten en un objeto del tipo derivada3, asi pues una expresion como

```
obj.i = 20;
```

hacia cual **i** se esta haciendo referencia? La que se encuentra en derivada1 o derivada2? Como existen dos copias de base presentes en el objeto 'obj' entonces hays dos obj.i's. Como puede ver, la declaracion es inherentemente ambigua.

Existen dos maneras de remediar el anterior programa. La primera es aplicar el operador de resolucion de ambito para manualmente seleccionar una 'i'. Por ejemplo, la siguiente version el programa compilara y se ejecutara como se esperaba:

```
// Este programa usa el la resolucion de
// ambito explicita para seleccionar 'i'
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derivada 1 hereda base
class derivada1 : public base {

public:
 int j;
};

// derivada2 hereda base
class derivada2 : public base {

public:
 int k;
};

/* derivada3 hereda de ambas, derivada1 y derivada2
   esto significa que hay 2 copias de base en derivada3!
 */
```

```
class derivada3 : public derivada1, public derivada2 {
public:

 int sum;
};

int main()
{

  derivada3 obj;

  obj.derivada1::i = 10; // ambito resuelto, se usa la 'i' derivada1
  obj.j = 20;
  obj.k = 30;

  // ambito resuelto
  obj.sum = obj.derivada1::i + obj.j + obj.k;

  // tambien resuelto aqui..
  cout << obj.derivada1::i << " ";

  cout << obj.j << " " << obj.k << " ";
  cout << obj.sum;

return 0;
}
```

Aplicando **::**, el programa manualmente selecciona a derivada1 como version de base. Sin embargo, esta solucion levanta un detalle profundo: Que pasa si solo una copia de base es en realidad requerida? Existe alguna forma de prevenir que dos copias sean incluidas en derivada3? La respuesta, como probablemente ha adivinado se logra con 'clases base virtuales'.

Cuando dos o mas objetos son derivados de una clase base común, puede prevenir múltiples copias de la clase base de estar presentes en un objeto derivado por esas clases, declarando la clase base como virtual cuando esta es heredada. para hacer esto, se precede el nombre de la clase base con la palabra virtual cuando esta es heredada.

Para ilustrar este proceso, aqui esta otra version del programa de ejemplo. Esta vez, derivada3 contiene solo una copia de base.

```
// Este programa usa clases base virtuales.
#include <iostream>
using namespace std;

class base {

public:
 int i;
};

// derivada1 hereda base como virtual
class derivada1 : virtual public base {

public:
```

```
 int j;
};

// derivada2 hereda base como virtual
class derivada2 : virtual public base {
public:
 int k;
};

/* derivada3 hereda ambas, derivada1 y derivada2.
   Esta vez, solo existe una copia de la clase base.
 */

class derivada3 : public derivada1, public derivada2 {
public:
 int sum;
};

int main()
{

  derivada3 obj;

  obj.i = 10;  // ahora no-ambiguo
  obj.j = 20;
  obj.k = 30;

  // no-ambiguo
  obj.sum = obj.i + obj.j + obj.k;

  // no-ambiguo
  cout << obj.i << " ";

  cout << obj.j << " " << obj.k << " ";
  cout << obj.sum;

return 0;
}
```

Como puede ver la palabra clave 'virtual' precede el resto de la especificación de la clase heredada. Ahora que ambas, derivada1 y derivada2 han heredado base como virtual, cualquier herencia multiple que las incluya causará que sólo una copia de base se encuentre presente. Además, en derivada3 existe sólo una copia de base, y obj.i = 10 es perfectamente válida y no-ambiguo.

Otro punto a tener en mente: incluso cuando ambas, derivada1 y derivada2 espcifican base como virtual, base sigue presente en los dos tipos derivados. Por ejemplo, la siguiente secuencia es perfectamente válida:

```
// Definir una clase del tipo derivada1

derivada1 miclase;

miclase.i = 88;
```

La diferencia entre una clase base normal y una virtual se hace evidente sólo cuando un objeto hereda la clase base mas de una vez; si la clase base ha sido declarada como virtual, entonces sólo una instancia de esta estará presente en el objeto heredado. De otra manera, múltiples copias serían encontradas.

← Sobrecarga de Operadores Introducción Funciones virtuales →