

Herencia y Composición en POO

Contenido

Introducción	1
Concepto de la Herencia.....	2
Relación de Herencia y diagrama de clases	2
Declaración de la Herencia en C++	3
Implantación de la Herencia en C++	4
Control de acceso en la herencia.....	8
Redefinición de miembros.....	9
Constructores y destructores en la herencia.....	11
Relación de Composición	12
Declaración de la Composición.....	13
Implementación de la Composición	13
Conclusión	17
Ejercicio	17

Introducción

Hay dos mecanismos para construir clases utilizando otras clases:

- Decir que una nueva clase es una especialización de una clase existente y que tiene unas cosas (atributos y métodos) más que la clase general: **Herencia**.
- Pegar objetos de otras clases dentro de una nueva clase: **Composición**.

Ejemplos de composición y herencia:

- Clase persona y clase empleado.
 - Herencia: un empleado es una persona.
- Clase persona y clase domicilio.
 - Composición: una persona tiene un domicilio.
- Clase empresa, clase empleado y clase jefe de grupo de empleados.
 - Herencia entre empleado y jefe: Un jefe es un empleado.
 - Composición entre empresa y empleado o jefe.

Estos son los mecanismos esenciales para evitar la repetición de código y permitir la reusabilidad, y por tanto la reutilización de software, que es una de las características más importantes de la programación orientada por Objeto. En C++, el efecto de la reutilización de código son similares tanto en la composición como en la herencia (lo cual tiene sentido, pues ambas son dos formas

de crear nuevos tipos utilizando tipos ya existentes). Sin embargo su uso, funcionamiento, declaración e implementación son diferentes.

Concepto de la Herencia

Es el proceso mediante el cual un objeto de una clase adquiere propiedades definidas en otra clase que lo preceda en una jerarquía de clasificaciones. Permite la definición de un nuevo objeto a partir de otros, agregando las diferencias entre ellos.

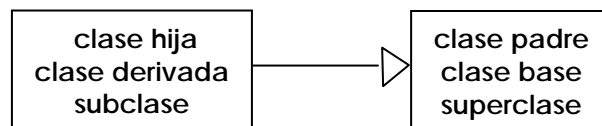
Relación de Herencia y diagrama de clases

La herencia (generalización/especialización) representa la relación "es un" o "es una" entre clases, por ejemplo:

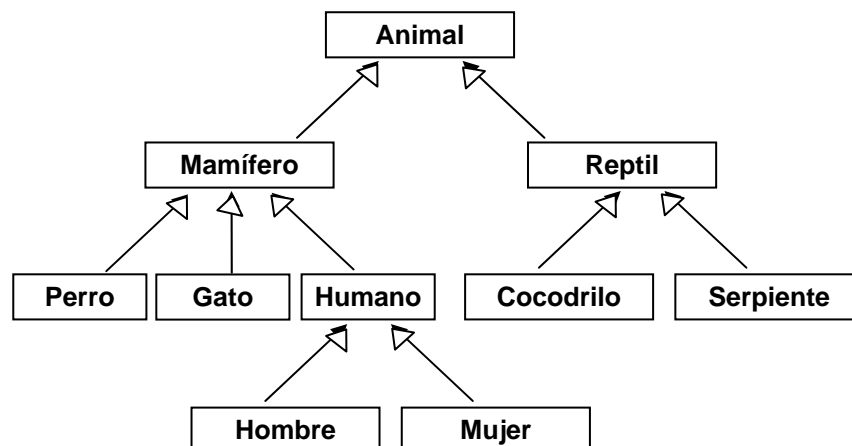
- Un profesor **es un** empleado, Un obrero **es un** empleado
- Un carro **es un** vehículo, Una bicicleta **es un** vehículo
- Un mamífero **es un** animal, Un ave **es un** animal

La herencia es un tipo de jerarquía de clases en la que cada subclase (clase hija, clase derivada) pueden acceder tanto a los atributos como a los métodos públicos y protegidos de la superclase (clase padre, clase base). Cada subclase o clase hija en la jerarquía es siempre una extensión de la(s) superclase(s) que además incorpora atributos y métodos propios.

En el diagrama de clases: La herencia se representa mediante una relación de generalización/especificación, que se denota con un triángulo de la siguiente forma:



Ejemplo: El siguiente diagrama de clases muestra la relación de Herencia entre la superclase Animal y sus subclases



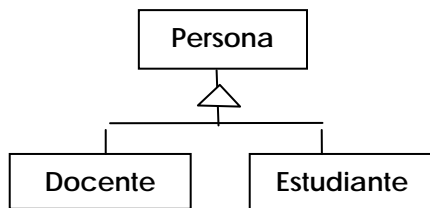
Las clases heredan los datos y métodos de la superclase. Un método heredado puede ser sustituido por uno propio si ambos tienen el mismo nombre.

Tipos de herencia

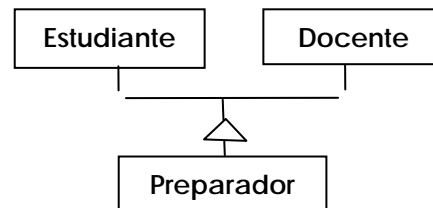
La herencia se clasifica según el modo de acceso a los miembros de la clase base:

- Pública (public): los modos de acceso a los miembros de la clase base se quedan igual en la clase derivada.
- Protegida (protected): los miembros "public" de la clase base pasan a ser "protected". El resto se queda igual.
- Privada (private): todos los miembros de la clase base pasan a ser "private" en la derivada.

La herencia también puede ser **simple** (cada clase tiene sólo una superclase) o **múltiple** (cada clase puede tener asociada varias superclases). La clase Docente y la clase Estudiante heredan las propiedades de la clase Persona (superclase, herencia simple). La clase Preparador (subclase) hereda propiedades de la clase Docente y de la clase Estudiante (herencia múltiple).



Herencia simple



Herencia múltiple

Declaración de la Herencia en C++

Declarar en C++ la herencia de las clases se realiza con el siguiente esquema:

```
class <clase_derivada>:<tipo> <lista de clase_base> {  
    //Datos miembros o atributos de la clase derivada  
    //Funciones miembros o métodos de la clase derivada  
}
```

Por ejemplo, según el diagrama anterior la clase **Docente** hereda de **Persona** (herencia simple):

```
class Docente: public Persona {  
    //Datos miembros o atributos de la clase Docente  
    //Funciones miembros o métodos de la clase Docente
```

```
}
```

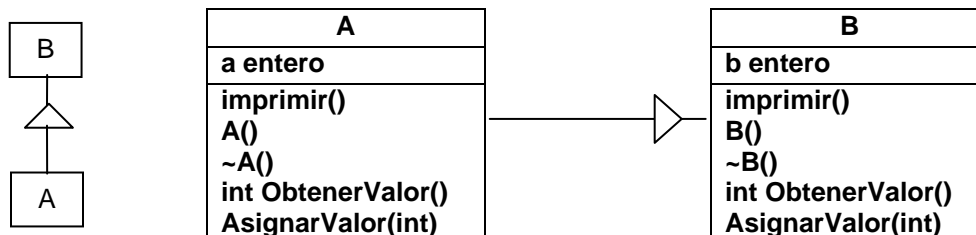
La clase **Preparador** hereda de **Docente** y **Estudiante** (herencia múltiple):

```
class Preparador: public Docente, Estudiante {  
    //Datos miembros o atributos de la clase Preparador  
    //Funciones miembros o métodos de la clase Preparador  
}
```

Cuando se hereda, realmente se expresa "Esta nueva clase (derivada) es como esta otra vieja clase (base)". Se comienza el código proporcionando el nombre de la clase, como se realiza normalmente, pero antes de abrir la llave del cuerpo de la clase, se colocan dos puntos y el nombre de la clase base (o de las clases bases, separadas por comas, para herencia múltiple). Una vez realizado, automáticamente se consiguen todos los miembros y las funciones de la clase base.

Implantación de la Herencia en C++

El siguiente diagrama de clases muestra una relación de herencia entre la clase A y clase B: "A es un B"



El código C++ esta organizado los siguientes archivos

- **B.h**: Archivo cabecera contentivo de la clase base B y los prototipos de los métodos.
- **B.cpp**: Archivo contentivo de la implantación de todos los métodos de la clase base B.
- **A.h**: Archivo cabecera contentivo de la clase derivada A y los prototipos de los métodos.
- **A.cpp**: Archivo contentivo de la implantación de todos los métodos de la clase derivada A.
- **Principal.cpp**: Archivo con la función principal (main) donde se harán algunas pruebas.

(1) Edite el archivo B.h y agregue el siguiente código C++:

```
// INICIO Archivo B.h  
/***** Declaración de la Clase base B*****/  
class B {  
    private:  
        int b;
```

```

public:
    B();
    ~B();
    int ObtenerValor();
    void AsignarValor(int);
    void imprimir();
};
// FIN archivo B.h

```

(2) Edite el archivo B.cpp y agregue el siguiente código C++:

```

// INICIO Archivo B.cpp
#include "B.h"
#include <iostream>
using namespace std;

/** Implementación de los métodos***/
B::B() { // Constructor: inicializa los atributos
    b = 6;
}

B::~B() {
}

int B::ObtenerValor() { // retorna el dato miembro b
    return b; //return this->b;
}

void B::AsignarValor(int V) {
    this->b = V; //asigna b = V;
}

void B::imprimir() { // imprime por consola el dato miembro b
    cout << "b = " << b << endl;
}
// FIN archivo B.cpp

```

Como se puede observar, las clases bases (clase B) no muestran en su declaración e implementación las clases derivadas (clase A). En cambio las clases derivadas si deben hacer notar la herencia (es decir mencionar las clases bases de las cuales hereda de forma inmediata, pues la herencia es transitiva). El código C++ de la clase derivada A con respecto a la clase B presenta la declaración de la herencia.

(3) Edite el archivo A.h y agregue el siguiente código C++:

```

// INICIO Archivo A.h
#include "B.h"
/** Declaración de la Clase derivada A que hereda de B ***/
class A : public B {

```

```

private:
    int a;
public:
    A();
    ~A();
    int ObtenerValor();
    void AsignarValor(int);
    void imprimir();
};
// FIN de archivo A.h

```

(4) Edite el archivo B.cpp y agregue el siguiente código C++:

```

// INICIO Archivo A.cpp
#include "A.h"
#include <iostream>
using namespace std;

/** Implementación ****/
A::A() { // Constructor: inicializa los atributos
    a = 3;
}

A::~A() {
}

int A::ObtenerValor() { // retorna el dato miembro valor
    return a; //return this->a;
}

void A::AsignarValor(int V) {
    this->a = V; //asigna a = V;
}

void A::imprimir() { // imprime por consola el dato miembro a
    cout << "a = " << a << endl;
    B::imprimir(); // imprime por consola el dato miembro b
                    // heredado de la clase B
}
// FIN de archivo A.cpp

```

(5) Edite el archivo Principal.cpp y coloque las siguientes instrucciones en código C++ para probar la herencia de B en la clase A:

```

/* Para correr este programa debe ejecutar el
* siguiente comando:
*
* ./AheredaB
*
*/

```

```

// INICIO archivo Principal.cpp
#include <iostream>
using namespace std;

#include "A.h"

int main()
{
    cout << "Imprimir los tamaños de las clases" << endl;
    cout << "sizeof(A) = " << sizeof(A) << endl;
    cout << "sizeof(B) = " << sizeof(B) << endl;
    // Declaración de 2 objetos, uno de cada clase
    cout << "Se instancias Objetos de la Clase A y B" << endl;
    A objA;
    B objB;
    objA.AsignarValor(55);
    objB.AsignarValor(112);
    cout << "Valor del dato miembro luego de la asignación = " <<
endl;
    cout << "A = " << objA.ObtenerValor() << " B = " <<
objB.ObtenerValor() << endl;
    cout << "Función Imprimir Clase A, objeto objA: " << endl;
    objA.imprimir();
    cout << "Función Imprimir Clase B, objeto objB: " << endl;
    objB.imprimir();
    cout << "Tamaño del objeto A = " << sizeof(objA) << endl << "
Tamaño del objeto B = " << sizeof(objB) << endl;
    return 0;
}
// FIN archivo Principal.cpp

```

Las intrucciones para correr este programa están en el comentario ubicado al principio del archivo Principal.cpp

(6) Por último edite un archivo de nombre Makefile y coloque las siguientes líneas dentro de él.

```

CC = g++
CFLAGS = -g
OTHERFLAGS = -c
main: Principal A B
$(CC) $(CFLAGS) Principal.o B.o A.o -o AheredaB
Principal: Principal.cpp
$(CC) $(CFLAGS) $(OTHERFLAGS) Principal.cpp
B: B.cpp
$(CC) $(CFLAGS) $(OTHERFLAGS) B.cpp
A: A.cpp
$(CC) $(CFLAGS) $(OTHERFLAGS) A.cpp

```

(7) 5. Para compilar ejecute el comando: make

La salida que muestra la ejecución del programa es la siguiente:

El compilador anuncia que b es un dato privado incluso en la clase A que lo hereda y que no es posible tener acceso aunque la herencia sea pública.

(2) Cambie la declaración de la clase A en A.h para heredar de forma protegida, compile y ejecute. En otras palabras realice el siguiente cambio:

```
class A : protected B {  
    ...  
};
```

En este caso el compilador da el mismo mensaje pues el dato miembro en B sigue siendo privado.

(3) Cambie la definición de B para que permita que al ser heredada se compartan sus datos miembros con acceso de lectura y escritura. Compile y ejecute. En otras palabras realice el siguiente cambio en B.h:

```
class B {  
    protected:  
        int b;  
    public:  
        . . .  
};
```

En esta oportunidad termina la compilación y además los métodos de la clase A tienen acceso directo al dato miembro b heredado. Si cambia el tipo de herencia de A nuevamente a public se obtiene el mismo resultado.

Cómo se observa, existen varios modos o tipos de herencia, es decir la manera como se realiza la herencia con el fin de controlar el acceso de lo que se hereda y que viene combinado con el tipo de acceso de los datos y funciones miembros. La siguiente tabla muestra esta relación:

	public	protected	private
Clase derivada	Accesible	Accesible	No Accesible
Fuera	Accesible	No Accesible	No Accesible

El tipo protected, solo tiene sentido para la herencia, lo que definimos como protected en la clase base es accesible en las clases derivadas, pero no es accesible desde fuera de las clases derivadas o base.

Redefinición de miembros

A veces, interesa cambiar en la subclase o clase derivada la definición de algo que está en la clase base. Por ejemplo en la clase A se puede agregar un atributo con el mismo nombre de un atributo de la clase base. Sin embargo, lo heredado no se pierde aunque se redefina, lo mismo pasa con los métodos.

- (1) Realice los siguientes cambios para redefinir el atributo b en la clase A:

En A.h:

```
class A : public B {
private:
    int a;
    int b[5];
public:
    . . .
};
```

En A.cpp

```
A::A() { // Constructor: inicializa los atributos
    a = 3;
    for (int i=0; i < 5; i++) b[i] = 0; // inicializa b
}

. . .

void A::imprimir() { // imprime por consola el dato miembro a
    cout << "a = " << a << endl;
    cout << "b[0] de A = " << b[0] << endl;
    cout << "b de B = " << B::b << endl;
}
```

Como se observa, el tamaño de A es mayor al agregarle más atributos a su definición. El atributo b de B y b de A forman parte del objeto A y se diferencian por el espacio de nombres (B::b en B y b en A) como lo muestra la función imprimir.

Los métodos redefinidos de la clase A no eliminan los métodos de la clase base, ellos siguen existiendo.

- (2) Compile y ejecute el siguiente cambio en el programa principal principal.cpp para que note como puede tener acceso desde el objeto A de un método de la clase base redefinido solo con indicar el espacio de nombre de la clase base:

```
    cout << "Función Imprimir Clase B, sobre el objeto objA: " <<
endl;
    objA.B::imprimir();
```

Constructores y destructores en la herencia

Construcción objeto clase derivada:

- Primero se construye la parte heredada de la clase(s) base: Se ejecutan constructores de las clases base.
- Por último se ejecuta el código del constructor de la clase derivada.

Destrucción objeto clase derivada:

- el proceso es a la inversa que en la construcción.
- Se ejecuta primero el destructor de la clase derivada y a continuación los de las clases base.

(1) Para probar esto basta con agregar en los realizar los siguientes cambios de los constructores y destructores:

En A.cpp

```
A::A() { // Constructor: inicializa los atributos
    a = 3;
    for (int i=0; i < 5; i++) b[i] = 0;
    cout << "* Constructor de la clase A * " << endl;
}

A::~~A() {
    cout << "* Destructor de la clase A * " << endl;
}
```

En B.cpp

```
B::B() { // Constructor: inicializa los atributos
    b = 6;
    cout << "* Constructor de la clase B * " << endl;
}

B::~~B() {
    cout << "* Destructor de la clase B * " << endl;
}
```

(2) Cambie, compile y ejecute el principal con lo siguiente:

```
#include <iostream>
using namespace std;
#include "A.h"
void prueba() { // probar ejecución de los constructores y
destructores
    A a;
    a.AsignarValor(5);
    cout << "Dentro de la funcion prueba" << endl;
}
int main()
{
    cout << "Imprimir los tamaños de las clases" << endl;
}
```

```

    cout << "sizeof(A) = " << sizeof(A) << endl;
    cout << "sizeof(B) = " << sizeof(B) << endl;
    prueba();
    return 0;
}

```

Note que la Salida muestra el orden de los constructores bases-derivada y es inverso en las llamadas de los destructores:

```

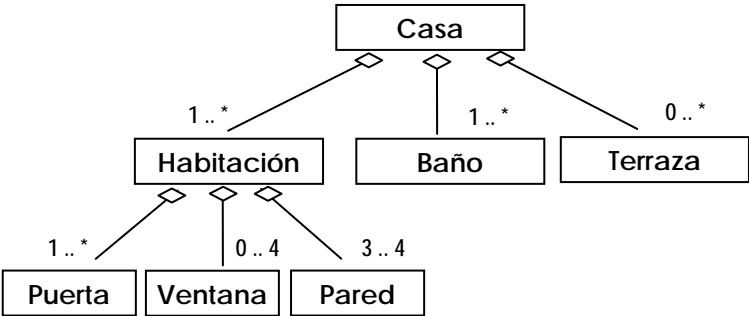
* Constructor de la clase B *
* Constructor de la clase A *
Dentro de la funcion prueba
* Destructor de la clase A *
* Destructor de la clase B *

```

Relación de Composición

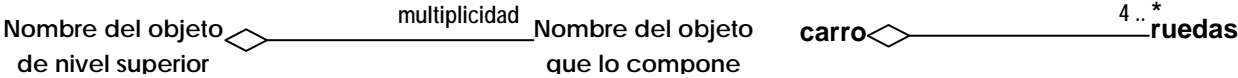
Es una relación que representa a objetos compuestos por otros objetos. El objeto en el nivel superior de la jerarquía es el todo y los que están en los niveles inferiores son sus partes o componentes. Un componente es parte esencial de una entidad. La relación es fuerte al punto que si el componente es eliminado o desaparece, la clase mayor (el todo) deja de existir.

Por ejemplo, la entidad Casa puede ser modelada en términos de sus componentes de la siguiente forma:



La relación de agregación/composición, se denota a continuación:

Agregación:



Composición:



Si la entidad motor forma parte de carro, y corazón forma parte de la persona, entonces la flecha apunta a la clase carro y persona respectivamente y el diamante va pegado a estas clases. La

diferencia entre la relación carro-ruedas y persona-corazón, es que en el primer caso se llama agregación por ser una relación en donde la existencia del carro no depende de las ruedas, mientras que en el segundo caso una persona no existe sin su corazón. Por tanto la relación de la persona con su corazón es de composición y se denota con el rombo relleno.

Declaración de la Composición

La Composición en C++ no es más que considerar la posibilidad de que los atributos o datos miembros de una clase sean objetos, es decir una clase puede tener como datos miembros a objetos de otras clases.

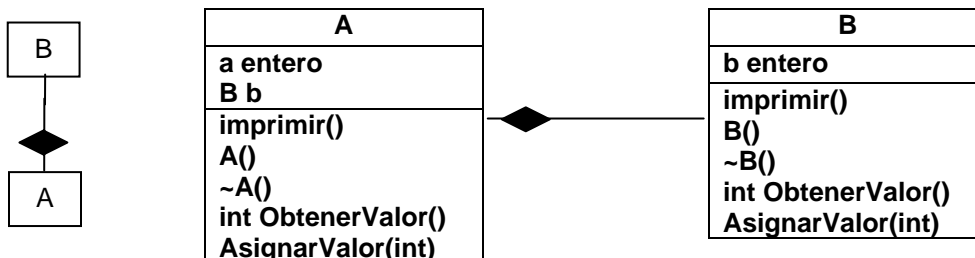
Por ejemplo, la clase empleado siguiente tiene 4 atributos: el nombre, apellido, fecha de nacimiento y fecha de contratación del empleado, se define de la siguiente forma:

```
// incluye la definición de la clase Fecha de fecha.h
#include "fecha.h"
class Empleado {
private:
    char nombres[ 25 ];
    char apellidos[ 25 ];
    const Fecha fechaNacimiento; // composición: objeto miembro
    const Fecha fechaContratacion; // composición: objeto miembro
public:
    Empleado(
        const char *, const char *, const Fecha &, const Fecha & );
    void imprime() const;
    ~Empleado(); // destructor
}; // fin de la clase Empleado
```

La clase Fecha debe existir para crear la clase Empleado y se relaciona con Empleado a través de una relación de composición al tener dos miembros del tipo Fecha.

Implementación de la Composición

El siguiente diagrama de clases muestra una relación de composición entre la clase A y clase B: "B es parte A" o "A esta compuesto por B".



El código C++ esta organizado los siguientes archivos

- **B.h**: Archivo cabecera contentivo de la clase B y los prototipos de los métodos.
- **B.cpp**: Archivo contentivo de la implantación de todos los métodos de la clase B.
- **A.h**: Archivo cabecera contentivo de la clase A y los prototipos de los métodos.
- **A.cpp**: Archivo contentivo de la implantación de todos los métodos de la clase A.
- **Principal.cpp**: Archivo con la función principal (main) donde se harán algunas pruebas.

- (1) Conserve el archivo B.h anterior
- (2) Conserve el archivo B.cpp anterior
- (3) Realice el cambio para que A tenga un dato miembro del tipo B, es decir para que B forme parte de A, copie el siguiente código:

```
// INICIO Archivo A.h
#include "B.h"
/** Declaración de la Clase A que esta formada por B ****/
class A {
    private:
        int a;
        B b;
    public:
        A();
        ~A();
        int ObtenerValor();
        void AsignarValor(int);
        void imprimir();
};
// FIN de archivo A.h
```

Y en A.cpp

```
// INICIO Archivo A.cpp
#include "A.h"
#include <iostream>
using namespace std;

/** Implementación ****/
A::A() { // Constructor: inicializa los atributos
    a = 3;
    cout << " Constructor de la clase A * " << endl;
}

A::~A() {
    cout << " Destructor de la clase A * " << endl;
}

int A::ObtenerValor() { // retorna el dato miembro valor
    return a; //return this->a;
```

```

}

void A::AsignarValor(int V) {
    this->a = V; //asigna a = V;
}

void A::imprimir() { // imprime por consola los datos miembro A
    cout << "a = " << a << endl;
    cout << "b de A = ";
    b.imprimir(); // método imprimir de la clase B.
}
// FIN de archivo A.cpp

```

Note los cambios en la manera de imprimir el atributo b de la clase A.

- (4) Compile y pruebe dichos cambios en el programa principal que se muestra a continuación y utilizando el siguiente makefile.

```

/* Para correr este programa debe ejecutar el
* siguiente comando:
*
* ./BesparteA
*
*/
// INICIO archivo Principal.cpp
#include <iostream>
using namespace std;

#include "A.h"

void prueba() { // probar ejecución de los constructores y destructores
    A a;
    a.AsignarValor(5);
    cout << "Dentro de la función prueba" << endl;
}

int main()
{
    cout << "Imprimir los tamaños de las clases" << endl;
    cout << "sizeof(A) = " << sizeof(A) << endl;
    cout << "sizeof(B) = " << sizeof(B) << endl;
    prueba(); // prueba de llamadas a los constructores

    cout << endl << "Se instancias Objetos de la Clase A y B" << endl;
    A objA;
    B objB;
    objA.AsignarValor(44);
    objB.AsignarValor(100);
    cout << "Valor del dato miembro luego de la asignación = " <<

```

```

endl;
    cout << "A = " << objA.ObtenerValor() << " B = " <<
objB.ObtenerValor() << endl;
    cout << "Función Imprimir Clase A, objeto objA: " << endl;
    objA.imprimir();
    cout << "Función Imprimir Clase B, objeto objB: " << endl;
    objB.imprimir();

    return 0;
}
// FIN archivo Principal.cpp

```

El makefile:

```

CC = g++
CFLAGS = -g
OTHERFLAGS = -c
main: Principal A B
$(CC) $(CFLAGS) Principal.o B.o A.o -o BesparteA
Principal: Principal.cpp
$(CC) $(CFLAGS) $(OTHERFLAGS) Principal.cpp
B: B.cpp
$(CC) $(CFLAGS) $(OTHERFLAGS) B.cpp
A: A.cpp
$(CC) $(CFLAGS) $(OTHERFLAGS) A.cpp

```

- (5) Para compilar ejecute el comando: make. La salida es la siguiente:

```

Imprimir los tamaños de las clases
sizeof(A) = 8
sizeof(B) = 4
* Constructor de la clase B *
* Constructor de la clase A *
Dentro de la funcion prueba
* Destructor de la clase A *
* Destructor de la clase B *

Se instancias Objetos de la Clase A y B
* Constructor de la clase B *
* Constructor de la clase A *
* Constructor de la clase B *
Valor del dato miembro luego de la asignacion =
A = 44 B = 100
Funcion Imprimir Clase A, objeto objA:
a = 44
b de A = b = 6
Funcion Imprimir Clase B, objeto objB:
b = 100
* Destructor de la clase B *
* Destructor de la clase A *
* Destructor de la clase B *

```

Note como este cambio de relación entre la clase A y B no cambia los siguientes detalles:

- a) El tamaño de los objetos de tipo A se mantiene constante, pues están compuestos por dos enteros y su tamaño es 8 byte. Por tanto, la creación del objeto A es la misma en ambas relaciones.
- b) La ejecución de los constructores y destructores de las clases es la misma, pues en el caso de la composición no se puede crear un objeto de la clase A antes de crear su miembro del tipo B, por tanto la secuencia de construcción es B,A.

Sin embargo, con la composición si se cambia la forma como se imprime los valores del dato miembro b. Cuando es heredado se imprime con las instrucciones

```
cout << "b de B = " << B::b << endl;
```

o con su método

```
B::imprimir(); // imprime por consola el dato miembro b
```

Pero al ser ahora una parte del objeto A por composición entonces, debe hacerse a través de su método

```
b.imprimir();
```

Conclusión

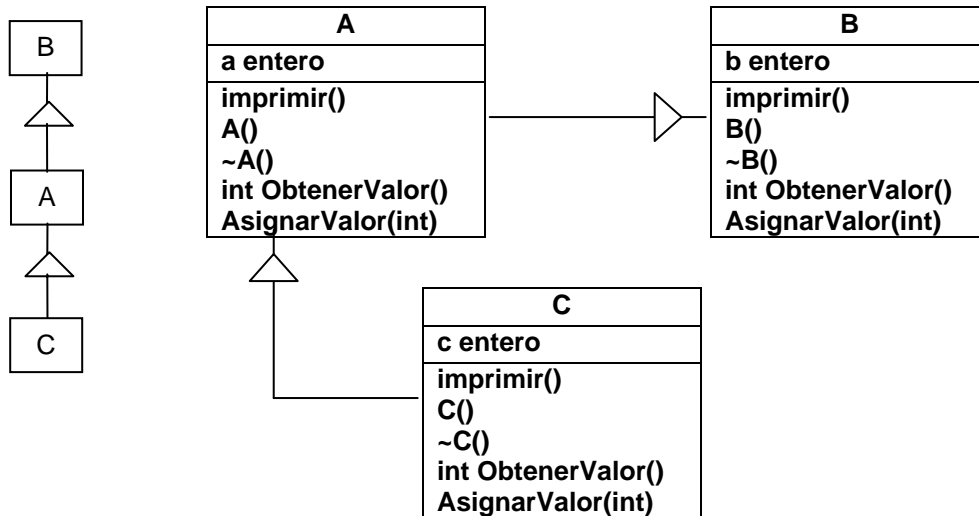
Las dos técnicas fundamentales para construir nuevas clases a partir de otras ya existentes son la herencia y la composición. En el diseño y programación orientada a objeto se debe decidir cual de estos dos tipos usar en las relaciones entre las clases que modelas las entidades del mundo real. Se debe comprender que ambos tipos de relaciones pueden existir al mismo tiempo en un problema en donde hay varias clases y diversos tipos de relaciones.

La composición de clases expresa el hecho de que se pueden componer o constituir clases nuevas a partir de objetos de otras clases. Lo mismo que en el mundo real, en donde observamos objetos formados por otros objetos: computadores formados por teclado, pantalla y unidad central de proceso; rectas formadas por puntos, vehículos formados por motor, transmisión y chasis, etc.

La herencia de clases es un concepto diferente al de la composición. La herencia permite construir una clase derivada a partir de una clase base. La clase derivada hereda todas las propiedades de la clase base, es decir, sus variables (datos), y sus funciones y operadores miembro. Todas las nuevas características que se quiere dar a la clase derivada se le confieren por medio de nuevas variables, y de nuevas funciones y operadores. En ambas técnicas, el código de las clases originales o clases bases no presenta ningún tipo de modificación (clase B).

Ejercicio

- (1) Agregar al primer diagrama de clases una clase C que hereda de A e implemente dicho cambio en su programa, modificando el makefile



- (2) Realice los cambios convenientes para que un objeto del tipo C pueda inicializar (en su constructor), cambiar, leer e imprimir todos sus datos miembros, los propios y heredados.
- (3) Realice los cambios convenientes para que las relaciones de herencia sean ahora de composición y se siga permitiendo que un objeto del tipo C pueda inicializar (en su constructor), cambiar, leer e imprimir todos sus datos miembros.

Elaborado: prof. Hilda Contreras

Revisado: prof. Rafael Rivas y prof. Gilberto Díaz.