**MARCO TEORICO DEL PARADIGMA ORIENTADO A OBJETOS**

**Y**

**EL LENGUAJE DE PROGRAMACIÓN C++**

**Conceptos Teóricos del Paradigma Orientado a Objetos**

- [Introducción](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Programación_orientada_a): Definición, abstracción, encapsulación, polimorfismo, herencia,

- [Programación Imperativa y Orientada a Objetos](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Programación_Imperativa_y)

- [Imperativa o Procedimental](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Imperativa_o_procedimental)

- [Orientada a Objetos](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Orientada_a_Objetos)

- [Lenguajes Orientados a Objetos](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Lenguajes_Orientados_a)

- [Clases](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Clases)

- [Abstracción](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Abstracción_en_programación)

- [Encapsulación](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Encapsulación_en_programación)

- [Polimorfismo](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Polimorfismo_en_programación)

- [Herencia](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_Herencia_en_programación)

**El Lenguaje de Programación C++ (Aplicaciones del Paradigma Orientado a Objetos)**

- [Introducción](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_INTRODUCCION) Conceptos de Encapsulamiento, Herencia, Polimorfismo

- [Definición de Clases](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_DEFINICIÓN_DE_CLASES) Introducción, Funciones Miembros, Constructores, Destructores, Ámbito de Clases, Especificardores de Accesos, Accesos Privilegiados.

**-** [Herencia](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_HERENCIA) Introducción, Derivación de Clases, Marcadores de Accesos

- [Manejo de clases en módulos](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_MANEJO_DE_CLASES)

- [Funciones virtuales](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_FUNCIONES_VIRTUALES)

- [Declaración de funciones virtuales](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_DECLARACION_DE_FUNCIONES)

- [Objetos dinámicos](http://www.frt.utn.edu.ar/sistemas/paradigmas/poo.htm#_OBJETOS_DINAMICOS)

**Introducción a la Programación Orientada a Objetos**

La programación **Orientada a Objetos (POO)** es una metodología de diseño de software y un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan *estado* (es decir, datos) y *comportamiento* (esto es, procedimientos o métodos). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que se comunican entre ellos para realizar tareas. Esto difiere de los lenguajes procedurales tradicionales, en los que los datos y los procedimientos están separados y sin relación. Estos métodos están pensados para hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.

Otra manera en que esto es expresado a menudo, es que la programación orientada a objetos anima al programador a pensar en los programas principalmente en términos de tipos de datos, y en segundo lugar en las operaciones ("métodos") específicas a esos tipos de datos. Los lenguajes procedurales animan al programador a pensar sobre todo en términos de procedimientos, y en segundo lugar en los datos que esos procedimientos manejan.

Los programadores que emplean lenguajes procedurales, escriben funciones y después les pasan datos. Los programadores que emplean lenguajes orientados a objetos definen objetos con datos y métodos y después envian mensajes a los objetos diciendo que realicen esos métodos en sí mismos.

Algunas personas también diferencian la POO sin clases, la cual es llamada a veces programación basada en objetos.

Hay un cierto desacuerdo sobre exactamente que características de un método de programación o lenguaje le califican como "orientado a objetos", pero hay un consenso general en que las características siguientes son las más importantes (cada una de las cuales se explica más detalladamente en su respectiva página enlazada):

**Abstracción**: Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar *cómo* se implementan estas características. Los procesos, las funciones o los métodos pueden tambien ser abstraídos y cuando los están, una variedad de técnicas son requeridas para ampliar una abstracción.

**Encapsulación**: También llamada "ocultación de la información", esto asegura que los objetos no pueden cambiar el estado interno de otros objetos de maneras inesperadas; solamente los propios métodos internos del objeto pueden acceder a su estado. Cada tipo de objeto expone una *interfaz* a otros objetos que especifica cómo otros objetos pueden interactuar con él. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción.

**Polimorfismo**: Las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del referente. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama *asignación tardía* o *asignación dinámica*. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.

**Herencia**: Organiza y facilita el polimorfismo y la encapsulación permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que reimplementar su comportamiento. Esto suele hacerse habitualmente agrupando los objetos en *clases* y las clases en *árboles* o *enrejados* que reflejan un comportamiento común.

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard del Centro de Cómputo Noruego en Oslo. Según se informa, la historia es que trabajaban en simulaciones de naves, y fueron confundidos por la explosión combinatoria de cómo las diversas cualidades de diversas naves podían afectar unas a las otras. La idea ocurrió para agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus *propios* datos y comportamiento. Fueron refinados más tarde en Smalltalk, que fue desarrollado en Simula en Xerox PARC pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "en marcha" en lugar de tener un sistema basado en programas estáticos.

La programación orientada a objetos tomó posición como la metodología de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de C++ , una extensión del lenguaje de programación C. Su dominación fue consolidada gracias al auge de las Interfaces gráficas de usuario, para los cuales la programación orientada a objetos está particularmente bien adaptada.

Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp, Pascal, y otros. La adición de estas características a los lenguajes que no fueron diseñados inicialmente para ellas condujo a menudo a problemas de compatibilidad y a la capacidad de mantenimiento del código. Los lenguajes orientados a objetos "puros", por otra parte, carecían de las características de las cuales muchos programadores habían venido depender. Para saltar este obstáculo, se hicieron muchas tentativas para crear nuevos lenguajes basados en métodos orientados a objetos, pero permitiendo algunas características procedurales de maneras "seguras". El Eiffel de Bertrand Meyer fue un temprano y moderadamente acertado lenguaje con esos objetivos pero ahora ha sido esencialmente reemplazado por Java, en gran parte debido a la aparición de Internet, para la cual Java tiene características especiales.

Apenas como programación procedural conducida a los refinamientos de la técnica, tales como la programación estructurada, los métodos modernos de diseño de software orientado a objetos incluyen refinamientos tales como el uso de los patrones de diseño, diseño por contrato, y lenguajes de modelado (tales como UML).

**Programación Imperativa y Orientada a Objetos**

**Imperativa o procedimental**

La programación procedimental clásica presenta ciertos problemas, que han ido haciéndose cada vez más graves, a medida que se construían aplicaciones y sistemas informáticos más complejos, entre los que destacan los siguientes:

Modelo mental anómalo. Nuestra imagen del mundo se apoya en los seres, a los que asignamos nombres sustantivos, mientras la programación clásica se basa en el comportamiento, representado usualmente por verbos.

Es difícil modificar y extender los programas, pues suele haber datos compartidos por varios subprogramas, que introducen interacciones ocultas entre ellos.

Es difícil mantener los programas. Casi todos los sistemas informáticos grandes tienen errores ocultos, que no surgen a la luz hasta después de muchas horas de funcionamiento.

Es difícil reutilizar los programas. Es prácticamente imposible aprovechar en una aplicación nueva las subrutinas que se diseñaron para otra.

**Orientada a Objetos**

La programación orientada a objetos (OOP, por las siglas inglesas de Object-Oriented Programming) es una nueva forma de programar que proliferó a partir de los años ochenta y trata de encontrar solución a estos problemas utilizando los siguientes conceptos:

**Objetos**: entidades complejas provistas de datos (propiedades, atributos) y comportamiento (funcionalidad, programas, métodos).Corresponden a los objetos reales del mundo que nos rodea.

**Clases**: conjuntos de objetos que comparten propiedades y comportamiento.

**Herencia**: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.

**Encapsulamiento**: cada objeto está aislado del exterior, es un módulo natural, y la aplicación entera se reduce a un agregado o rompecabezas de objetos. El aislamiento protege a los datos asociados a un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

**Polimorfismo**: programas diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, aunque el significado del programa varíe según el objeto al que se aplica.

La programación orientada a objetos introduce nuevos conceptos, que a veces no son más que nombres nuevos aplicados a conceptos antiguos, ya conocidos. Entre ellos destacan los siguientes:

**Método**: es un programa asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena mediante un "mensaje".

**Mensaje**: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros.

**Propiedad, atributo o variable**: datos asociados a un objeto o a una clase de objetos.

En la programación orientada a objetos pura no deben utilizarse llamadas de subrutinas, únicamente mensajes.

Por ello, a veces recibe el nombre de **programación sin CALL**, igual que la programación estructurada se llama también **programación sin GOTO**.

Sin embargo, no todos los lenguajes orientados a objetos prohíben la instrucción CALL (o su equivalente), permitiendo realizar programación híbrida, procedimental y orientada a objetos a la vez.

**Lenguajes Orientados a Objetos:**

Smalltalk, Objective-C, C++, Ada 95, Java, Python

Delphi, Lexico (en castellano), C Sharp, Eiffel

Ruby, ActionScript, Visual Basic.NET, PHP

Estos lenguajes de programacion son muy avanzados en orientacion a objetos.

Al igual que C++ otros lenguajes, como OOCOBOL, OOLISP, OOPROLOG y OOREXX, han sido creados añadiendo extensiones orientadas a objetos a un lenguaje de programación clásico.

**Clases**

Se puede pensar en una **clase** como en un molde para un objeto. a partir de la clase, se pueden crear o instancear objetos con propiedades especificas la instanceacion crea un objeto con valores específicos para cada una de las propiedades.

La clase más general se denomina *clase base* o *súper clase*. Debajo de esta clase base se encuentran las subclases o clases derivadas que heredan mensajes y propiedades de sus ancestros.

**Abstracción en programación orientada a objetos**

Consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el "¿qué hace?" mas que en el "¿cómo lo hace?". El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos ha sido el nivel de abstracción del que cada uno de ellos hace uso.

**Encapsulación en programación orientada a objetos**

En programación orientada a objetos, se denomina **encapsulación** al ocultamiento del estado, es decir, de los datos miembro, de un objeto de manera que sólo se puede cambiar mediante las operaciones definidas para ese objeto.

De esta forma el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

**Polimorfismo en programación orientada a objetos**

En programación orientada a objetos se denomina **polimorfismo** a la propiedad que poseen algunas operaciones de tener un comportamiento diferente dependiendo del objeto sobre el que se aplica.

El polimorfismo dinámico puede consistir en que si se tiene una instancia de una subclase, se asigna esa instancia a una referencia de la clase madre y se llama a un método de la referencia se obtendrá el comportamiento de la subclase. El mecanismo concreto depende del lenguaje de programación

**Herencia en programación orientada a objetos**

La herencia es uno de los mecanismos de la programación orientada a objetos, por medio de la cual una clase se deriva de otra de manera que extiende su funcionalidad. Una de sus funciones más importantes es la de proveer polimorfismo y *late binding*.

Tipos de herencia

Herencia simple: Un objeto puede extender las características de otro objeto y de ningún otro, es decir, solo puede tener un padre.

Herencia múltiple: Un objeto puede extender las características de uno o más objetos, es decir, puede tener varios padres. En este aspecto hay discrepancias entre los diseñadores de lenguajes. Algunos de ellos han preferido no admitir la herencia múltiple por las posibles coincidencias en nombres de métodos o datos miembros. Por ejemplo C++ admite herencia múltiple, Java y Ada sólo herencia simple.

**LENGUAJE DE PROGRAMACIÓN C++**

**INTRODUCCIÓN**

Para enfrentarse a las complejidades del mundo real, el ser humano ha desarrollado la capacidad de generalizar, clasificar y generar abstracciones. Así, tenemos un vocabulario donde muchos de los sustantivos representan clases de objetos.  
Los objetos de cada clase comparten ciertos atributos o rasgos de comportamiento.

Por ejemplo, al tratarse algún tema relacionado con las aves no es necesario referirnos a alguna característica de una ave en particular sino a los atributos que comparten todas ellas.  
Los lenguajes de Programación Orientada a Objetos pueden dividirse en dos grupos:  
El primero formado por los llamados lenguajes "puros" (llamados así debido a que sólo permiten programar con el paradigma de la Programación Orientada a Objetos) dentro del cual se encuentran SmallTalk y Actor, entre otros.  
El segundo grupo lo forman los lenguajes que permiten, además de la Programación Orientada a Objetos, la programación procedimental, razón por la que se les llama "híbridos". A este grupo pertenece el lenguaje C++.  
Para el manejo de los lenguajes de Programación Orientada a Objetos es necesario estudiar los conceptos de :

ENCAPSULAMIENTO  
HERENCIA  
POLIMORFISMO

En las siguientes secciones se estudia detalladamente cada una de ellas.

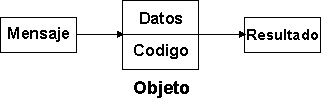
La Programación Orientada a Objetos es un método de programación que intenta imitar la forma en que modelamos el mundo real y tiene como meta principal mejorar la eficiencia y productividad de los grupos de programadores. Muchos expertos afirman que la Programación Orientada a Objetos ayuda a reducir la complejidad, especialmente en programas grandes ( de 10,000 líneas o más ), además de que permite a los programadores reutilizar el código existente en lugar de reescribir módulos que ya trabajan correctamente.

**ENCAPSULAMIENTO**

El encapsulamiento consiste en poner juntos los datos y las funciones dentro de un objeto de tipo clase.  
El modelo procedimental puede representarse como en la figura.

http://www.frt.utn.edu.ar/sistemas/ico/Tutorial%20C++1_archivos/t82_archivos/mem20.gif

- Modelo procedimental.  
  
donde se observa que los datos y el código se manejan como partes separadas. El programa ( código ) es alimentado con los datos para que produzca resultados.  
Por otra parte, el modelo orientado a objetos puede representarse como en la figura 8.2.



- Modelo de la Programación Orientada a Objetos.  
  
donde DATOS y CODIGO se han unido para formar un OBJETO, el cual va a producir ciertos RESULTADOS, de acuerdo al MENSAJE que se le envíe.   
Esto es, un objeto de ciertas características va a comportarse de acuerdo a su clase al recibir un mensaje específico.   
Como se logra el ENCAPSULAMIENTO en el lenguaje C++ ?   
Podemos pensar que, así como se requieren tipos para definir las variables en un enfoque procedimental, se requieren "tipos" o "moldes" para crear los objetos.  
  
En el lenguaje C++ esos moldes se crean por medio de las palabras reservadas : class, struct y union .  
La Geometría representa un campo idóneo para ejemplificar el manejo de objetos. Las definiciones para cada una de las figuras se hacen a través de enunciados que incluyen la expresión: " .. es el lugar geométrico de todos los puntos que ...". Como puede verse, el concepto de punto es la base para la definición de cualquier figura geométrica.   
La representación de las figuras geométricas, en la pantalla de un monitor, se hace por medio de los pixeles, que son los elementos básicos ( puntos iluminados ) utilizados para representar los gráficos en la pantalla.  
Con base a estos conceptos, en la siguiente sección vamos a crear una clase base llamada punto.  
Los objetos de tipo punto podrán situarse en cualquier lugar de la pantalla utilizando un par de coordenadas ( x,y ).

**DEFINICIÓN DE CLASES**

La definición de la clase punto en C++ puede tomar la siguiente forma :

*class punto*

*{*

*int x,y ; // MIEMBROS DATO*

*} ;*

Aquí no se está haciendo uso del encapsulamiento, puesto que no se ha declarado ninguna función miembro dentro de la clase

**FUNCIONES MIEMBRO**

En otros lenguajes orientados a objetos, a las funciones miembro se les conoce como métodos .   
Para incluir una función miembro en la definición de una clase, existen dos formas :

1.- Definir la función dentro de la definición de la clase.  
Por ejemplo :

*class punto*

*{*

*int x,y ;*

*int dax() { return x ; } // FUNCION EN LINEA*

*};*

2.- Declarar la función dentro de la definición de la clase y escribir la definición de la función fuera de la definición de la clase.  
Por ejemplo :

*class punto*

*{*

*int x,y ;*

*int dax() ; // DECLARACION DE LA FUNCION MIEMBRO*

*};*

*int punto::dax() // DEFINICION DE LA FUNCION MIEMBRO*

*{*

*return x ;*

*}*

En la línea de cabecera de la definición de la función miembro dax() se utiliza el operador de resolución de ambito :: . Este operador indica que la función dax() es una función miembro de la clase punto.

**CONSTRUCTORES**

Los constructores y los destructores son dos tipos especiales de funciones miembro.  
  
Un constructor especifica la manera en que será creado e inicializado un nuevo objeto de cierta clase.  
Los constructores en C++ pueden ser definidos por el usuario ó generados por el lenguaje.  
El compilador de C++ invoca automáticamente al constructor apropiado cada vez que se defina un nuevo objeto.  
Esto puede ocurrir en una declaración de datos, cuando se copia un objeto o a través de la asignación dinámica de memoria a un nuevo objeto por medio del operador new.  
Para añadir un constructor a la clase punto escribimos :

*class punto*

*{*

*int x,y ;*

*int dax() { return x ; }*

*int day() { return y ; }*

*punto(int nx, int ny) ; // DECLARACION DEL CONSTRUCTOR*

*};*

*punto::punto(int nx, int ny) // DEFINICION DEL CONSTRUCTOR*

*{*

*x = nx ;*

*y = ny ;*

*}*

**DESTRUCTORES**

Los destructores destruyen a los objetos creados, liberando la memoria asignada. Pueden ser invocados explícitamente por medio del operador delete.  
  
Siguiendo con el ejemplo de la clase punto :

*class punto*

*{*

*int x,y ;*

*int dax() { return x ; }*

*int day() { return y ; }*

*punto(int nx, int nx) ;*

*~punto() ; // DECLARACION DEL DESTRUCTOR*

*};*

*punto::punto(int nx, int nx)*

*{*

*x = nx ;*

*y = ny ;*

*}*

*punto::~punto() // DEFINICION DEL DESTRUCTOR*

*{*

*delete x ;*

*delete y ;*

*}*

**AMBITO DE CLASES**

El término ámbito se refiere al área dentro de un programa donde un identificador dado es accesible.  
Para controlar la manera en que son accesados los miembros de las clases, C++ ha introducido el concepto de ámbito de clase . Todos los miembros de una clase están en el ámbito de esa clase, de manera que cualquier miembro de una clase puede referenciar a cualquier otro miembro de la misma clase.  
Las funciones miembro de una clase tienen acceso irrestricto a los miembros dato de esa misma clase. El acceso a los datos y funciones miembro de una clase exterior al ámbito de la clase es controlado por el programador a través de los especificadores de acceso.

**ESPECIFICADORES DE ACCESO**

Para el control de acceso a los miembros de una clase, se cuenta con tres palabras clave : private: , protected: y public: .  
La palabra clave apropiada ( incluyendo los dos puntos ) debe colocarse antes de las declaraciones que van a afectarse.

*private: Los miembros que siguen pueden ser accesados solamente*

*por las funciones miembro declaradas dentro de la*

*misma clase.*

*protected: Los miembros que siguen pueden ser accesados por*

*funciones miembro declaradas dentro de la misma clase,*

*y por funciones miembro de clases derivadas de esta*

*clase.*

*public: Los miembros que siguen pueden ser accesados desde*

*cualquier lugar dentro del mismo ámbito de la*

*definición de la clase.*

Por predeterminación, los miembros de las clases de tipo class son private , los de las clases de tipo struct son public.  
Por ejemplo, para hacer que las funciones miembro de la clase punto puedan ser accesadas desde cualquier lugar, escribimos :

*class punto*

*{*

*int x, y ; // privadas POR PREDETERMINACION*

*public: // LO QUE SIGUE ES público*

*int dax() { return x; }*

*int day() { return y; }*

*punto ( int nx, int ny ) ;*

*~punto();*

*}*

*punto::punto(int nx, int ny)*

*{*

*x = nx ;*

*y = ny ;*

*}*

*punto::~punto()*

*{*

*delete x ;*

*delete y ;*

*}*

**ACCESO PRIVILEGIADO**

Una clase puede darle a otra clase o función acceso privilegiado a sus áreas privadas. Tal acceso debe darse explícitamente, declarando la otra clase o función por medio del modificador friend (amigo). Los amigos son tratados como si fueran miembros de la clase y tienen acceso irrestricto a las áreas privadas de los objetos.  
Por ejemplo :

*class cosa*

*{*

*private:*

*int dato ;*

*public:*

*friend void carga( cosa t , int x ) ;*

*};*

*void carga ( cosa t , int x )*

*{*

*t.dato = x ;*

*}*

La definición de la clase cosa contiene el prototipo para la función carga y la diseña como un amigo. La función carga no es un miembro de la clase cosa, sino que es una función independiente a la que se le ha dado acceso especial a los miembros privados de los objetos de tipo cosa.  
Debido a que es un amigo de la clase cosa, carga puede accesar directamente los datos miembro privados de su parámetro t.  
Una clase completa puede hacerse un amigo de otra, tal como en:

*class perico ;*

*class paloma*

*{*

*public:*

*friend class perico ;*

*};*

*class perico*

*{*

*//CUALQUIER COSA*

*};*

*#include <iostream.h>*

*class punto*

*{*

*int x,y ;*

*public:*

*punto(int nx, int ny) { x = nx ; y = ny ; }*

*int dax() { return x ; }*

*int day() { return y ; }*

*};*

*void main()*

*{*

*int datox, datoy ;*

*cout << "COORDENADA EN X : " ; cin>> datox ;*

*cout << "COORDENADA EN y : " ; cin>> datoy ;*

*punto objeto(datox , datoy) ;//INVOCA AL CONSTRUCTOR punto*

*cout << "X=" << objeto.dax() ;*

*cout << '\n' ;*

*cout << " Y=" << objeto.day() ;*

*cout << '\n' ;*

*}*

*</pre>*

*</B></I><h4>Listado 8.1.- Utilización de la clase punto.</h4>*

*<TABLE BORDER=0 WIDTH=" 99%">*

**HERENCIA**

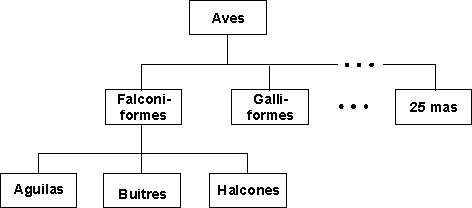
En la Programación Orientada a Objetos, la herencia es un concepto semejante al de la herencia genética que se dá en la naturaleza.  
En el reino animal, por ejemplo, podemos hablar de una clase base llamada aves, cuyos miembros poseen:

*- plumas*

*- alas*

*- pico*

Sin embargo, no todas las aves son iguales, por lo que se tiene que recurrir a las clases derivadas para distinguir unas aves de otras, como se muestra en la figura 8.3.



Jerarquía de clases a partir de la clase aves.

Cada una de las clases derivadas conserva las características de la clase base aves, pero posee algunas otras que la distingue de las demás clases.   
Hasta aquí, hemos hablado solamente de clases de aves no de las aves en sí. No nos hemos referido a ninguna ave en particular, por lo que pudieramos decir que hemos estado hablando de los "tipos de aves".  
Una vez que se tienen los tipos, podemos llevar a cabo la construcción de aves. Cuando tales aves hayan cumplido su misión, podremos destruirlas.  
Esto último puede parecer demasiado cruel en el mundo animal, pero en el caso de la Programación Orientada a Objetos es lo más práctico, pues resultaría un gasto innecesario de memoria el conservar los objetos que ya no van a utilizarse .  
Las clases, al ser consideradas como tipos, pueden utilizarse para construir bloques de programa. Si un tipo no cumple con las especificaciones de un nuevo programa, se puede utilizar para crear otro nuevo. Este nuevo tipo se modifica para usarse en la elaboración de nuevos objetos, y a su vez queda disponible para elaborar otros tipos, según se necesite.  
Es en esta creación de nuevos tipos donde se llega a la utilización de los modelos jerárquicos de clasificación.  
En C++, la herencia se maneja por medio de la derivación de clases.

**DERIVACIÓN DE CLASES**

La derivación de clases en C++ se rige por la siguiente sintaxis :

*class derivada : modificador\_de\_acceso base*

*{*

*.....*

*.....*

*}*

modificador\_de\_acceso es opcional y puede ser public o private.

**MODIFICADORES DE ACCESO**

Los modificadores de acceso se utilizan para cambiar la accesibilidad de los miembros heredados de acuerdo a lo que se observa en la tabla 8.1 :

|  |  |  |
| --- | --- | --- |
| EN CLASE BASE | MODIF. DE ACCESO | EN CLASE DERIVADA |
| private | private | inaccesible |
| protected | private | private |
| public | private | private |
| private | public | inaccesible |
| protected | public | protected |
| public | public | public |

Tabla 8.1.- Modificadores de acceso para las clases derivadas.  
  
Cuando se utiliza la palabra class, el modificador de acceso está predefinido como private ; cuando se utiliza la palabra struct, está predefinido como public.  
Al crear nuevas clases que dependen de clases existentes, debe asegurarse de que entiende la relación existente entre las clases base y derivada. Para esto es vital el entendimiento de los niveles de acceso conferidos por los especificadores private , protected y public .  
El C++ permite pasar los derechos de acceso de padres a hijos sin exponer los datos a quienes no son miembros ni amigos de la familia.   
El nivel de acceso de un miembro de una clase base, no es necesariamente el mismo cuando se maneja en el ámbito de la clase base que cuando se maneja en el ámbito de la clase derivada. El programador tiene el control sobre los niveles de acceso en todos los ámbitos.  
Como puede observarse en la tabla 8.1, el especificador private convierte los miembros public y protected de la clase base a miembros private en la clase derivada. Los miembros con especificador private en la clase base, mantienen ese especificador en la clase derivada.  
Aunque en las clases definidas por medio de la palabra class el nivel de acceso esta predeterminado como private, conviene explicitar los niveles de cada uno de los miembros de cada clase, para evitar confusiones.  
Por lo tanto, puede escribirse :

*class ejemplo*

*{*

*private:*

*int num ;*

*float costo ;*

*public:*

*void valida( char \* ) ;*

*int precio( float ) ;*

*};*

Una clase derivada hereda todos los miembros de su clase base, pero puede utilizar solamente los miembros protegidos y públicos de su clase base.  
Los miembros privados de la clase base no están disponibles directamente a través de los miembros de la clase derivada.  
Al usar la derivación public, los miembros protegidos de la clase base permanecen protegidos en la clase derivada; así que no estarán disponibles desde el exterior, excepto para otras clases derivadas públicamente y para los miembros y clases amigos.

**MANEJO DE CLASES EN MODULOS**

Las clases pueden empacarse en módulos para su utilización posterior en el desarrollo de programas.  
Una clase es inherentemente modular con sus miembros dato, funciones miembro y control de acceso. En el desarrollo de un programa, a veces tiene sentido poner las declaraciones, para cada clase o grupo relacionado de clases, por separado en un archivo de cabecera; y en otro las definiciones para las funciones miembro que no estén en línea.  
En el listado 8.2 se presenta el archivo de cabecera PIXEL.H , donde, a partir de la clase base punto, se deriva otra clase llamada pixel que nos servirá para visualizar u ocultar puntos en cualquier lugar de la pantalla.

*///////////////////////////////////////////////////////////*

*// PIXEL.H : CONTIENE LAS CLASES : //*

*// //*

*// punto QUE MANEJA LA LECTURA Y COLOCACION //*

*// DE UN PUNTO. //*

*// //*

*// pixel MANEJA LA VISIBILIDAD DE UN PUNTO EN //*

*// LA PANTALLA. //*

*///////////////////////////////////////////////////////////*

*enum booleano { falso, verdadero } ;*

*class punto*

*{*

*protected: //PERMITE QUE LAS CLASES DERIVADAS HEREDEN*

*//DATOS privados.*

*int x , y ;*

*public: //LAS SIGS.FUNCIONES PUEDEN ACCESARSE DESDE*

*//EL EXTERIOR.*

*punto(int nx, int ny) ;*

*int dax();*

*int day();*

*};*

*// CLASE DERIVADA DE punto. LA DERIVACION public SIGNIFICA*

*// QUE x , y ESTARAN PROTEGIDAS DENTRO DE pixel.*

*class pixel : public punto*

*{*

*protected: //LAS CLASES DERIVADAS DE pixel NECESITAN*

*//ACCESO A ESTOS MIEMBROS DATO.*

*booleano visible ;*

*public:*

*pixel(int nx, int ny); // CONSTRUCTOR*

*void mostrar();*

*void ocultar();*

*boleano es\_visible();*

*void mover\_hacia(int nuevax, int nuevay);*

*};*

Listado 8.2.- Archivo PIXEL.H  
  
Nótese la utilización de protected en lugar de private para ciertos elementos, con lo que se hace posible la utilización del archivo PIXEL.H en aplicaciones que manejen clases derivadas de punto y pixel.  
Las definiciones para las funciones miembro de las clases punto y pixel se almacenan en el archivo PUNTO2.CPP del listado 8.3.

*/////////////////////////////////////////////////////////////*

*// PUNTO2.CPP : CONTIENE LA DEFINICION PARA LAS FUNCIONES //*

*// MIEMBRO DE LAS CLASES punto y pixel //*

*/////////////////////////////////////////////////////////////*

*#include "pixel.h"*

*#include <graphics.h>*

*// DEFINICION DE LAS FUNCIONES MIEMBRO PARA LA CLASE punto*

*punto::punto(int nx , int ny)*

*{*

*x = nx ;*

*y = ny ;*

*};*

*int punto::dax(void)*

*{*

*return x ;*

*};*

*int punto::day(void)*

*{*

*return y ;*

*};*

*// DEFINICION DE LAS FUNCIONES MIEMBRO PARA LA CLASE pixel*

*pixel::pixel(int nx, int ny) : punto(nx,ny)*

*{*

*visible = falso ;*

*};*

*void pixel::mostrar(void)*

*{*

*visible = verdadero ;*

*putpixel(x,y, getcolor()); // USA EL COLOR PREDEFINIDO.*

*// SUPONE QUE EN main() SE*

*// INICIALIZAN LOS GRAFICOS.*

*};*

*void pixel::ocultar(void)*

*{*

*visible = falso ;*

*putpixel(x,y,getbkcolor()); // USA EL COLOR DEL FONDO.*

*};*

*booleano pixel::es\_visible(void)*

*{*

*return visible ;*

*};*

*void pixel::mover\_hacia(int nuevax, int nuevay)*

*{*

*ocultar(); // HACE INVISIBLE AL PUNTO ACTUAL*

*x = nuevax ; // CAMBIA LAS COORDENADAS x,y .*

*y = nuevay ;*

*mostrar(); // MUESTRA EL PUNTO EN LA NUEVA LOCALIDAD.*

*};*

Listado 8.3.- Definición de las clases punto y pixel.  
  
Observe que, para definir el constructor de la clase derivada, se hace referencia al constructor de la clase base como en :

*pixel::pixel(int nx,intny) : punto(nx,ny)*

Esto significa que, para construir un objeto de la clase pixel, primero se invoca al constructor de la clase base punto con los argumentos nx y ny creando e inicializando los miembros dato x,y. Posteriormente se invoca al constructor pixel, creando e inicializando el miembro dato visible.  
  
Note que la referencia al constructor de la clase base aparece en la definición, no en la declaración, del constructor de la clase derivada.  
Si no se ha definido un constructor para una clase x, el C++ generará un constructor predefinido de la forma :

*x::x();*

esto es, un constructor sin argumentos.  
El listado 8.4 contiene un programa que muestra la utilización de la clase pixel .

*////////////////////////////////////////////////////////////*

*// PIXEL.CPP : APLICACION ELEMENTAL DE LA CLASE pixel //*

*// //*

*////////////////////////////////////////////////////////////*

*#include <graphics.h>*

*#include <conio.h>*

*#include "pixel.h"*

*int main()*

*{*

*// INICIALIZACION DEL SISTEMA DE GRAFICOS*

*int manej = DETECT, modo ;*

*initgraph ( &manej, &modo, "c:\\borlandc\\bgi");*

*// ENCIENDE, CAMBIA Y APAGA PIXELES*

*pixel unpixel(100,50) ; // DEFINE unpixel CON VALORES*

*// INICIALES x=100, y=50*

*unpixel.mostrar(); // unpixel SE ENCIENDE*

*getch() ; // ESPERA UNA PULSACION DE TECLA*

*unpixel.mover\_hacia(300,150); // MUEVE unpixel A 300,150*

*getch();*

*unpixel.ocultar(); // OCULTA A unpixel*

*getch();*

*closegraph();*

*return 0 ;*

*}*

Listado 8.4.- Utilización de la clase pixel.  
  
Como podrá observarse, no se ha incluido el archivo PUNTO2.CPP, por lo que este programa deberá compilarse utilizando la opción PROJECT del Ambiente Integrado de Desarrollo ( IDE ) del C++ de Borland, creando un archivo de proyecto PIXEL.PRJ que incluya los archivos GRAPHICS.H, PUNTO2.CPP y PIXEL.CPP .  
Uno de los atractivos de las clases es que pueden utilizarse para construir otras clases que respondan a las necesidades de una nueva aplicación.  
En el ejemplo que se muestra en el listado 8.5 se hace uso de la clase pixel para derivar una nueva clase llamada circulo, la cual contendrá funciones miembro para mostrar, ocultar, mover, expander y contraer circulos.

*/////////////////////////////////////////////////////////////*

*// CIRCULO.CPP : MANEJO DE LA CLASE circulo DERIVADA DE LA //*

*// CLASE pixel //*

*/////////////////////////////////////////////////////////////*

*#include <graphics.h>*

*#include <conio.h>*

*#include "pixel.h"*

*class circulo : pixel // DERIVADA PRIVADAMENTE DE pixel*

*{*

*int radio ;*

*public:*

*circulo(int nx, int ny, int nradio);*

*void mostrar(void);*

*void ocultar(void);*

*void expander(int cantexp);*

*void contraer(int cantcont);*

*void mover\_hacia(int nvax, int nvay);*

*};*

*circulo::circulo(int nx,int ny,int nradio) : pixel(nx,ny)*

*{*

*radio = nradio;*

*};*

*void circulo::mostrar(void)*

*{*

*visible = verdadero;*

*circle(x,y,radio); // DIBUJA EL CIRCULO*

*};*

*void circulo::ocultar(void)*

*{*

*unsigned int colortemp ; // PARA GUARDA EL COLOR ACTUAL*

*colortemp = getcolor(); // GUARDA EL COLORR ACTUAL*

*setcolor(getbkcolor()); // CAMBIA AL COLOR DEL FONDO*

*visible = falso ;*

*circle(x,y,radio); // DIBUJA EL CIRCULO CON EL COLOR*

*// DEL FONDO PARA BORRARLO*

*setcolor(colortemp); // RESTABLECE EL COLOR ORIGINAL*

*};*

*void circulo::expander(int cantexp)*

*{*

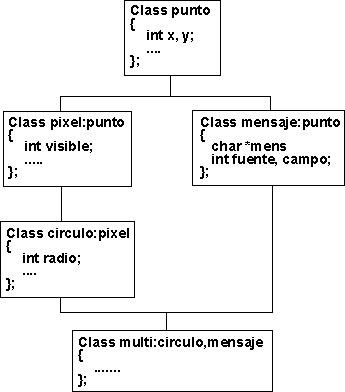
*ocultar(); // BORRA EL CIRCULO ACTUAL*

*radio += cantexp; // EXPANDE EL RADIO*

*if( radio <0 ) // EVITA RADIOS NEGATIVOS radio="0" ; mostrar(); // DIBUJA EL NUEVO CIRCULO }; void circulo::contraer(int cantcontr) { expander(-cantcontr); }; void circulo::mover\_hacia(int nvax, int nvay) { ocultar(); // BORRA EL CIRCULO ACTUAL x="nvax" ; // ASIGNA NUEVAS COORDENADAS y="nvay" ; mostrar(); // DIBUJA EL CIRCULO EN LA NUEVA POSICION }; void main() { int manej="DETECT" , modo ; initgraph(&manej, &modo, "c:\\borlandc\\bgi"); circulo uncirculo(100,200,50); uncirculo.mostrar(); getch(); uncirculo.mover\_hacia(200,250); getch(); uncirculo.expander(50); getch(); uncirculo.contraer(75); getch(); closegraph(); }*

**HERENCIA MULTIPLE**

Todos los lenguajes de Programación Orientada a Objetos manejan la herencia en la forma que hemos visto hasta aquí, donde cada clase derivada tiene sólo una clase base. Sin embargo, algunos de ellos permiten manejar lo que se llama herencia múltiple, en la cual se permite que una clase derivada pueda tener varias clases base.  
El mecanismo de herencia múltiple fué una de las principales características añadidas a la versión 2.0 de C++.  
  
Como un ejemplo del manejo de la herencia múltiple, consideremos el problema de tener que desplegar texto dentro de un círculo.  
Al principio podemos estar tentados a añadir una cadena de caracteres como miembro dato a la clase circulo y después agregarle código a circulo::mostrar para que despliegue el circulo con texto adentro.   
Pero el texto y el círculo son dos cosas diferentes. Cuando se piensa en texto, se piensa en tamaños y tipos de caracteres y en otros atributos que nada tienen que ver con los círculos.  
Sin embargo, podemos derivar una nueva clase directamente de la clase circulo y de otra clase que permita darle capacidad de manejar texto.   
En la figura 8.4 se ilustra este enfoque, el cual se detalla en el programa MULTI.CPP del listado 8.6.



*////////////////////////////////////////////////*

*// MULTI.CPP : ILUSTRA LA HERENCIA MULTIPLE //*

*////////////////////////////////////////////////*

*#include <graphics.h>*

*#include <conio.h>*

*#include <string.h>*

*#include "pixel.h"*

*class circulo : public pixel*

*{*

*protected:*

*int radio;*

*public:*

*circulo(int nx, int ny, int nradio) ;*

*void mostrar(void);*

*};*

*class mensaje : public punto // DESPLIEGA UN MENSAJE*

*{*

*char \*mens ; // MENSAJE A DESPLEGAR*

*int fuente ; // FUENTE A UTILIZAR*

*int campo ; // TAMAÑO DEL CAMPO PARA EL TEXTO*

*public:*

*mensaje( int mensx, int mensy, int mensf, int mensc,*

*char \*texto );*

*void mostrar(void); // MUESTRA EL MENSAJE*

*};*

*class multi : circulo, mensaje // HEREDA DE AMBAS*

*{ // CLASES.*

*public:*

*multi(int mcircx, int mcircy, int mcircr, int fuente,*

*char \*mens);*

*void mostrar(void); // MUESTRA CIRCULO CON MENSAJE*

*};*

*// DEFINICION DE LAS FUNCIONES MIEMBRO PARA LA CLASE circulo*

*circulo::circulo(int nx,int ny, int nradio) : pixel(nx,ny)*

*{*

*radio = nradio ;*

*};*

*void circulo::mostrar(void)*

*{*

*visible = verdadero ;*

*circle(x,y,radio) ; // DIBUJA EL CIRCULO*

*};*

*// DEFINICION DE LAS FUNCIONES MIEMBRO PARA LA CLASE mensaje*

*mensaje::mensaje(int mensx, int mensy, int mensf,*

*int mensc, char \*texto )*

*: punto(mensx,mensy)*

*{*

*fuente = mensf ;*

*campo = mensc ;*

*mens = texto ; // APUNTA A MENSAJE*

*};*

*void mensaje::mostrar(void)*

*{*

*int tam = campo / ( 8 \* strlen(mens)) ; // 8 PIXELES POR*

*// CARACTER*

*settextjustify(CENTER\_TEXT, CENTER\_TEXT);// CENTRA TEXTO*

*settextstyle(fuente, HORIZ\_DIR, tam) ; //AGRANDA SI tam >1*

*outtextxy(x,y, mens) ; // DESPLIEGA MENSAJE*

*};*

*// DEFINICION DE LAS FUNCIONES MIEMBRO PARA LA CLASE multi*

*multi::multi(int mcircx, int mcircy, int mcircr,*

*int fuente, char \*mens)*

*: circulo(mcircx, mcircy, mcircr) ,*

*mensaje(mcircx, mcircy, fuente,*

*2\*mcircr, mens)*

*{*

*};*

*void multi::mostrar(void)*

*{*

*circulo::mostrar();*

*mensaje::mostrar();*

*};*

*// DEFINICION DE LA FUNCION PRINCIPAL*

*void main()*

*{*

*int manej = DETECT, modo ;*

*initgraph(&manej, &modo, "c:\\borlandc\\bgi");*

*multi uno(250,100,25,SANS\_SERIF\_FONT,"Tu");*

*uno.mostrar();*

*multi dos(250,150,100,TRIPLEX\_FONT,"Tierra");*

*dos.mostrar();*

*multi tres(250,250,225,GOTHIC\_FONT,"Universo");*

*tres.mostrar();*

*getch();*

*closegraph();*

*}*

**POLIMORFISMO**

Las funciones miembro son las que determinan el comportamiento de un objeto. Esto conlleva la idea de que, si queremos que un objeto se comporte de manera diferente, primero habrá que modificar la clase o "molde", agregándole la función miembro que determina el comportamiento deseado, para, posteriormente, crear un objeto de la clase modificada.  
En los ejemplos mostrados hasta aquí, todos los objetos tienen la habilidad necesaria para mostrarse en la pantalla. Sin embargo, cada tipo de objeto difiere de los demás en la forma de mostrarse.  
Por ejemplo, un pixel puede mostrarse con sólo indicarle las coordenadas x,y correspondientes a su posición de la pantalla. En cambio, un círculo requiere de la medida del radio además de las coordenadas x,y de su centro.  
Esto nos lleva a pensar que para otros objetos tales como: lineas, rectangulos, elipses, etc. , pueden derivarse nuevas clases tomando como base la clase pixel y en las cuales se conservan los miembros dato x,y además de las funciones miembro dax() y day() definidos en la clase punto ; pero deben definirse nuevas funciones miembro mostrar(), así que, cada vez que se defina una nueva clase derivada de pixel, debe compilarse un nuevo proyecto ( archivo .PRJ ), donde se incluyan los archivos PUNTO2.CPP junto con el archivo correspondiente a la nueva figura ( por ejemplo LINEA.CPP ) para obtener el correspondiente archivo ejecutable ( por ejemplo LINEA.EXE ).  
Ya sea que hubieramos escrito las funciones mostrar() de todas las clases de figuras en un solo archivo fuente, o que las hubieramos manejado en varios archivos de un proyecto; habríamos tenido que volver a compilar cada vez que una nueva clase hubiera sido añadida.  
Suponiendo que ya se tienen las funciones mostrar() para cada uno de los objetos involucrados, ahora surge la pregunta:  
¿ A cuál de las funciónes mostrar() se está haciendo referencia ?   
Para responder a esto, con los mecanismos del C++ estudiados hasta antes de esta unidad, se pueden utilizar las siguientes formas:

*1.- La que realiza la distinción por medio del nombre ampliado.*

*Por ejemplo, mostrar(int,char) no es lo mismo que*

*mostrar(int,float).*

*2.- La que utiliza el operador de resolución de ámbito, por medio*

*del cual circulo::mostrar() se distingue de pixel::mostrar()*

*y de ::mostrar().*

*3.- La que distingue de acuerdo a la clase de objeto que se maneja.*

*Por ejemplo: uncirculo.mostrar() invoca a circulo::mostrar(),*

*mientras que unpixel.mostrar() invoca a pixel::mostrar().*

Todas esas resoluciones de función se han hecho en tiempo de compilación, por medio del mecanismo conocido como enlace temprano o estático.  
La alternativa en C++ se dá a través del polimorfismo, que consiste en la capacidad que posee el código para comportarse de diversas maneras. Cada forma de comportamiento dependerá de una situación específica, determinada en tiempo de ejecución, por medio del enlace tardío.  
El enlace tardío o dinámico trabaja por medio de funciones miembro especiales llamadas funciones virtuales.

**FUNCIONES VIRTUALES**

El concepto clave en las funciones virtuales es que las invocaciones a ellas se resuelven en tiempo de ejecución (de ahí el término enlace tardío). En nuestro ejemplo, esto significa que la invocación a la función mostrar() adecuada va a realizarse hasta que se conozca el tipo de objeto involucrado. Consideremos la función miembro circulo::mover\_hacia() definida en CIRCULO.CPP :

*void circulo::mover\_hacia(int nvax, int nvay)*

*{*

*ocultar();*

*x = nvax;*

*y = nvay;*

*mostrar();*

*}*

Observamos esta definición es parecida a la de pixel::mover\_hacia() que se encuentra en PUNTO2.CPP :

*void pixel::mover\_hacia(int nvax, int nvay)*

*{*

*ocultar();*

*x = nvax;*

*y = nvay;*

*mostrar();*

*}*

El valor de retorno, el nombre de la función, el número y tipo de argumentos, y hasta el cuerpo de la función, son idénticos.  
Si el C++ encuentra dos llamadas a función que tengan el mismo nombre de función pero diferentes nombres ampliados, el compilador resuelve las ambigüedades.  
Pero nuestras dos funciones mover\_hacia() , a simple vista, no ofrecen al compilador ninguna pista distintiva. ¿ Cómo va a saber el compilador a cuál función llamar ?. Cuando se trata de funciones miembro ordinarias, el compilador determina la función objetivo a partir de la clase del objeto involucrado en la llamada.   
Entonces, ¿ porqué no dejar que circulo herede la función mover\_hacia() desde pixel, tal como hereda las funciones miembro dax() y day(), desde la clase punto vía la clase pixel ?.  
La razón para no hacerlo es que un pixel se mueve de manera diferente a como se mueve un círculo y las funciones ocultar() y mostrar() invocadas en circulo::mover\_hacia() no son las mismas que se invocan en pixel::mover\_hacia(). Heredar mover\_hacia() desde pixel puede llevar a invocar a las funciones ocultar() y mostrar() erróneas cuando se intente mover un círculo.   
La solución a este problema es declarar a las funciones ocultar() y mostrar() como funciones virtuales. Esto retardará el enlace para que se hagan las llamadas a las versiones correctas de esas funciones cuando se invoque a la función mover\_hacia() ya sea para mover un pixel o para mover un círculo (o cualquier otra figura ).

**DECLARACION DE FUNCIONES VIRTUALES**

Para declarar una función como virtual, simplemente hay que anteponer la palabra virtual en la primera declaración de la función miembro.  
Por ejemplo :

*virtual void mostrar();*

*virtual void ocultar();*

Es importante recordar que solamente las funciones miembro pueden ser declaradas como virtual , y que no deben redeclararse en las clases derivadas.   
Si redeclaramos a la función miembro mostrar() con los mismos parámetros y el mismo tipo de retorno, en una clase derivada, la nueva función mostrar() automáticamente se vuelve virtual, ya sea que se utilice o no el calificador virtual. Se dice que la nueva función virtual mostrar() sobreescribe a la función mostrar() de la clase base.  
Tenemos la libertad de redeclarar a la función mostrar() con parámetros diferentes ( cambiando o no el tipo de retorno ), pero entonces el mecanismo virtual será inoperante para esta versión de mostrar().  
Al principio, deberá evitarse la sobrecarga temeraria, ya que hay situaciones en que una función no virtual puede ocultar a una función virtual en su clase base.  
A manera de ejemplo, vamos a crear un módulo ( Por medio de los archivos FIGURAS.H y FIGURAS.CPP mostrados más adelante ) que define algunas clases de figuras y crea un mecanismo generalizado para arrastrarlas a través de la pantalla.  
El principal objetivo, en el diseño del módulo, es permitir a los usuarios del mismo extender las clases definidas en él.  
Al añadir una nueva función miembro a una jerarquía de clases existente, surge la siguiente reflexión : ¿ En qué lugar de la jerarquía debe ser colocada la nueva función miembro ?  
La función arrastrar() debe ser un miembro de la clase pixel, para que todos los objetos de clases derivadas de pixel puedan arrastrase a través de la pantalla.  
Una vez decidido el lugar donde va a colocarse la función arrastrar(), vamos a rediseñar su definición.  
Vamos a re-escribir la función arrastar() para que las funciones que invoca, tales como: dax(), day(), mostrar(), mover\_hacia() y ocultar() , referencien las versiones correspondientes al tipo de objeto que va a ser arrastrado.  
En la nueva definición de la clase pixel, del archivo FIGURAS.H mostrado en el listado 8.7, se hacen virtuales a las funciones miembro mostrar() , ocultar() y arrastrar().

*////////////////////////////////////////////////////////*

*// FIGURAS.H : CONTIENE LA DEFINICION DE LAS CLASES //*

*// punto, pixel y circulo //*

*// EN UN SISTEMA A COMERCIALIZAR, ESTA SERIA LA //*

*// UNICA PARTE QUE DEBE DISTRIBUIRSE EN FORMA DE //*

*// CODIGO FUENTE. //*

*////////////////////////////////////////////////////////*

*enum booleano { falso, verdadero } ;*

*class punto*

*{*

*protected:*

*int x, y ;*

*public:*

*punto(int nx, int ny) { x = nx ; y = ny ; }*

*int dax() { return x ; }*

*int day() { return y ; }*

*};*

*class pixel : public punto*

*{*

*protected:*

*booleano visible ;*

*public:*

*pixel(int nx, int ny);*

*virtual void mostrar();*

*virtual void ocultar();*

*virtual void arrastrar(int narrastre);*

*booleano es\_visible() { return visible ; }*

*void mover\_hacia(int nvax, int nvay);*

*};*

*class circulo : public pixel*

*{*

*protected:*

*int radio;*

*public:*

*circulo(int nx, int ny, int nradio);*

*void mostrar();*

*void ocultar();*

*void expander(int cantexp);*

*void contraer(int cantcont);*

*};*

*// PROTOTIPO PARA LA FUNCION NO MIEMBRO leedelta()*

*// DEFINIDA EN EL LISTADO 8.8*

*booleano leedelta(int& deltax, int& deltay);*

El archivo FIGURAS.CPP, mostrado en el listado 8.8, contiene las definiciones para las funciones miembro declaradas en el listado 8.7.

*////////////////////////////////////////////////////////////*

*// FIGURAS.CPP : CONTIENE LAS DEFINICIONES PARA LAS //*

*// FUNCIONES DE LAS CLASES DECLARADAS EN FIGURAS.H //*

*// //*

*// EN UN CONTEXTO COMERCIAL, ESTE ARCHIVO PUEDE DISTRI- //*

*// BUIRSE EN FORMA DE OBJETO ( .OBJ ). //*

*////////////////////////////////////////////////////////////*

*#include <graphics.h>*

*#include <conio.h>*

*#include "figuras.h"*

*// DEFINICION DE LAS FUNCIONES MIEMBRO PARA LA CLASE pixel*

*pixel::pixel(int nx, int ny) : punto(nx,ny)*

*{*

*visible = falso ;*

*}*

*void pixel::mostrar()*

*{*

*visible = verdadero;*

*putpixel(x,y,setcolor());*

*}*

*void pixel::ocultar()*

*{*

*visible = falso;*

*putpixel(x,y, getbkcolor());*

*}*

*void pixel::mover\_hacia(int nvax, int nvay)*

*{*

*ocultar();*

*x = nvax ; y = nvay ;*

*mostrar();*

*}*

*void pixel::arrastrar(int narrastre)*

*{*

*int deltax, deltay ;*

*int figx, figy ;*

*mostrar() ; // DESPLIEGA LA FIGURA A SER ARRASTRADA*

*figx = dax();*

*figy = day();*

*// ESTE ES EL CICLO DE ARRASTRE*

*while(leedelta(deltax,deltay))*

*{*

*figx += (deltax \* narrastre) ;*

*figy += (deltay \* narrastre) ;*

*mover\_hacia(figx, figy) ;*

*};*

*}*

*// LA SIGUIENTE ES UNA FUNCION DE PROPOSITOS GENERALES QUE*

*// UTILIZA LAS TECLAS DE FLECHAS PARA MOVER EL CURSOR.*

*booleano leedelta(int& deltax, int& deltay)*

*{*

*char car;*

*booleano salir;*

*deltax = 0 ;*

*deltay = 0 ;*

*do{*

*car = getch() ; // LEE LA TECLA*

*if(car == 13) // RETORNO DE CARRO*

*return falso ;*

*if(car == 0 ) // CODIGO EXTENDIDO*

*{*

*salir = verdadero ; // ASUME QUE ES UTILIZABLE*

*car = getch() ; // TOMA EL RESTO DEL CODIGO*

*switch(car)*

*{*

*case 72 : deltay = -1 ; break ; // ARRIBA*

*case 80 : deltay = 1 ; break ; // ABAJO*

*case 75 : deltax = -1 ; break ; // IZQ.*

*case 77 : deltax = 1 ; break ; // DER.*

*default : salir = falso ; // OTRA*

*};*

*}*

*} while( !salir ) ;*

*return verdadero;*

*}*

*// DEFINICION DE LAS FUNCIONES MIEMBRO PARA LA CLASE circulo*

*circulo::circulo(int nx, int ny, int nradio) : pixel(nx, ny)*

*{*

*radio = nradio;*

*}*

*void circulo::mostrar()*

*{*

*visible = verdadero;*

*circle(x,y,radio);*

*}*

*void circulo::ocultar()*

*{*

*unsigned int colortemp;*

*colortemp = getcolor();*

*setcolor(getbkcolor());*

*visible = falso;*

*circle(x,y, radio);*

*setcolor(colortemp);*

*}*

*void circulo::expander(int cantexp)*

*{*

*ocultar();*

*radio += cantexp;*

*if(radio <0) radio="0;" mostrar(); } void circulo::contraer(int cantcont) { expander(-cantcont); }*

Ahora vamos a probar las nuevas clases punto, pixel y circulo utilizándolas para derivar una clase llamada arco que se encuentra definida en el archivo POLIMORF.CPP del listado 8.9.

*////////////////////////////////////////////////////////////*

*// POLIMORF.CPP : CONTIENE LA DEFINICION DE LA CLASE arco //*

*// QUE SE DERIVA DE circulo. //*

*// Para compilar, forme un proyecto con FIGURAS.OBJ y //*

*// POLIMOFIRSMO //*

*////////////////////////////////////////////////////////////*

*#include <graphics.h>*

*#include <conio.h>*

*#include "figuras.h"*

*class arco : public circulo*

*{*

*int anginic, angfin ;*

*public:*

*arco(int nx, int ny, int nradio,*

*int nanginic, int nangfin) : circulo(nx, ny, nradio)*

*{*

*anginic = nanginic;*

*angfin = nangfin;*

*}*

*void mostrar() ; // ESTAS FUNCIONES SON VIRTUALES EN*

*// pixel*

*void ocultar() ;*

*};*

*// DEFINICION DE LAS FUNCIONES MIEMBRO PARA LA CLASE arco*

*void arco::mostrar()*

*{*

*visible = verdadero ;*

*arc( x, y, anginic, angfin, radio ) ;*

*}*

*void arco::ocultar()*

*{*

*int colortemp;*

*colortemp = getcolor();*

*setcolor(getbkcolor());*

*visible = falso ;*

*arc(x, y, anginic, angfin, radio);*

*setcolor(colortemp);*

*}*

*// DEFINICION DE LA FUNCION PRINCIPAL*

*void main()*

*{*

*int manej =DETECT, modo ;*

*initgraph( &manej, &modo, "c:\\borlandc\\bgi");*

*circulo uncirculo(151,82,50);*

*arco unarco(151,82,25,0,190);*

*unarco.arrastrar(10) ; // LA FUNCION arrastar() SE APLICA*

*// A UNA FIGURA QUE NO CONOCE.*

*unarco.ocultar();*

*uncirculo.arrastrar(10);*

*closegraph();*

*getch();*

*}*

**OBJETOS DINAMICOS**

En los ejemplos mostrados anteriormente se utilizaron objetos cuyo espacio de almacenamiento se manejó en la pila.  
En esta sección trataremos con objetos dinámicos, cuyo espacio de almacenamiento en el montículo se asigna y se libera en tiempo de ejecución por medio de los operadores new y delete.   
Por ejemplo, si existe una clase llamada punto, podemos escribir:

*punto\* ap ;*

*ap = new punto(10,20);*

En la primera línea se está declarando un apuntador llamado ap, que servirá para apuntar a objetos de clase punto .  
En la segunda línea se gestiona un bloque de memoria lo suficientemente grande para almacenar un objeto de la clase punto, se invoca al constructor punto pasándole los argumentos 10,20 . La dirección de inicio del bloque gestionado se asigna al apuntador ap.  
Las dos líneas anteriores pueden sustituirse por:

*punto\* ap = new punto(10,20);*

En el listado 8.10 se presenta un ejemplo del manejo de dos objetos dinámicos, donde, además, se utiliza el polimorfismo.  
En 8.10 puede observarse que se declara el apuntador ap para que apunte a objetos de clase numero y, sin embargo, ese mismo apuntador se utiliza para apuntar a objetos de clase division.  
Esto se debe a que, en C++, un apuntador declarado para apuntar a objetos de una clase base puede utilizarse para apuntar a objetos de una clase derivada.  
Esta regla no es válida en el sentido inverso; esto es, un apuntador declarado para apuntar a objetos de una clase derivada no puede utilizarse para apuntar a objetos de la clase base, por lo que:

*division\* ap;*

*ap = new numero(10);*

produce un error.

*///////////////////////////////////////////////////////*

*// OBJDIN.CPP : Muestra el uso de objetos dinámicos. //*

*///////////////////////////////////////////////////////*

*#include <iostream.h>*

*class numero*

*{*

*protected:*

*int valor;*

*public:*

*numero(int numdado) { valor = numdado;}*

*virtual int davalor(){return valor;}*

*};*

*class division : public numero*

*{*

*protected:*

*int divisor;*

*public:*

*division(int d, int n) : numero(d) {divisor=n;}*

*virtual int davalor() {return valor/divisor;}*

*};*

*void main()*

*{*

*numero\* ap;*

*ap = new numero(15);*

*cout << ap->davalor() << '\n'; //Despliega 15 delete ap; ap="new" division(15,2); cout << ap->davalor() << '\n'; // Despliega 7 }*