40
2ej

Universidad Nacional Autónoma de México

FACULTAD DE CIENCIAS

Programación Orientada a Objetos en C++

T E S I S

QUE PARA OBTENER EL TITULO DE

A C T U A R I O

P R E S E N T A

MA. DE JESUS MADERA JARAMILLO

MEXICO, D.F. 1993

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

I N D I C E

INTRODUCCION

La programación orientada a objetos (POO) ha adquirido gran importancia desde hace pocos años y las definiciones de los conceptos más importantes de ésta son muchos y muy variados ya que cada autor las define desde su punto de vista y de acuerdo al lenguaje de programación orientado a objetos (LPOO) con el que trabaje.

El objetivo de este trabajo es dar una definición de los conceptos más comunes de la programación orientada a objetos, libre de la influencia de algún lenguaje en particular y luego detallar tales conceptos en el lenguaje de programación orientado a objetos C++.

La aplicación de los conceptos de la POO se hace en el LPOO C++ porque es uno de los lenguajes que más ha impulsado este modelo de programación, debido principalmente a que está basado en el lenguaje de programación C, el cual es ampliamente aceptado por ser muy portable.

El capítulo 1 contiene una breve historia de cómo han evolucionado las técnicas y lenguajes de programación. En el capítulo 2 se presentan las definiciones más comunes de la POO, y por último, en el capítulo 3 se explica como se aplican tales conceptos en el LPOO C++.

EVOLUCION DE LOS LENGUAJES Y TECNICAS DE PROGRAMACION

La evolución de los lenguajes y técnicas de programación se debe a las necesidades y reqerimientos de la industria.

Con el único lenguaje que se contaba para programar, en la década de los 40, era con el lenguaje de máquina. Después de esta etapa, los lenguajes y técnicas de programación han evolucionado desde el lenguaje ensamblador en los 50, a lenguajes orientados a procedimientos en los 60, programación estructurada y abstracción de datos en los 70, y para los 80 la programación orienlada a objetos, distribuida, funcional y relacional.

1.1 LENGUAJES DE PROGRAMACION

La clasificación genealógica de los lenguajes y técnicas de programación se muestra a continuación.

	Década	*Hardware*
1a Generación	40 y 50	Bulbos
Lenguaje de máquina		
Lenguaje ensamblador		
2a Generación	50 y 60	Transistores
FORTRAN, COBOL, ALGOL 58,		
ALGOL 60, LISP		

3a Generación	Finales de los	Circuitos
PL/I, ALGOL 68, Pascal y Simula	60 hasta los 90	Integrados

Brocha Generacional 70

Durante esta etapa aparecieron muchos lenguajes, entre ellos C, ADA, Smalltalk, CLU

Paradigmas de Lenguajes de Programación 80

Programación orientada a objetos, Distribuida, Funcional y Relacional.

Los Lenguajes de primera generación son los lenguajes de máquina y ensambladores. Los lenguajes de máquina están diseñados de manera que cada proposición sea interpretada directamente por la computadora, en otras palabras, el procesador es capaz de comprender cada proposición y realizar las operaciones correspondientes. Para expresar los algoritmos en lenguaje de máquina se necesitan muchas proposiciones, ya que éstas son muy simples y sólo expresan una parte muy pequeña del mismo, además las instrucciones están escritas en binario; es por esto que la programación en lenguaje de máquina es muy tediosa.

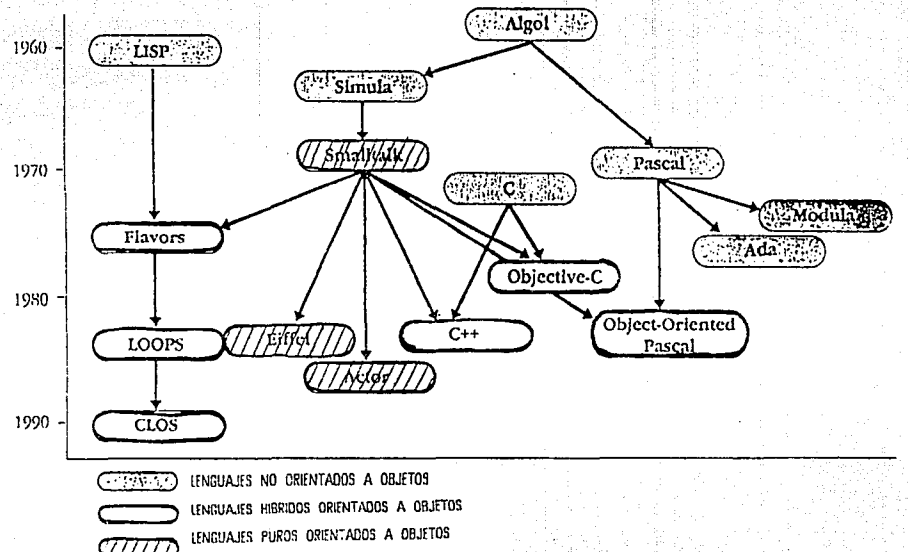
Los lenguajes ensambladores son ligeramente superiores a los

lenguajes de máquina. Un lenguaje ensamblador es específico a la computadora para la cual se diseñó, es decir, los programas escritos en un lenguaje ensamblador no pueden transferirse a otra computadora sin volver a escribirlos.

Los conceptos que se desarrollaron con los lenguajes de segunda generación fueron evaluación de expresiones aritméticas, declaraciones de variables y creación de arreglos, listas, pilas y subrutinas. Estos conceptos se mejoraron con los lenguajes de tercera generación. FORTRAN es un lenguaje para cálculo numérico, COBOL se aplica en programas administrativos, LISP es uno de los lenguajes más importantes en el área de la inteligencia artificial. ALGOL 60 no fue muy difundido pero tuvo mucha influencia en Pascal, PL/1, Simula y ADA.

En PL/1 se combinaron los elementos de FORTRAN, ALGOL Y COBOL. de ésta combinación resultó un lenguaje muy poderoso pero complejo y tan difícil de aprender como de implantar. En ALGOL 68 se generalizó ALGOL 60; ALGOL 68 es un lenguaje muy elegante pero no muy aceptado. Pascal es una extensión de ALGOL 60 y fue ampliamente aceptado por la comunidad académica, pero insuficiente para uso industrial.

Los lenguajes que más han influido en la programación orientada a objetos son: LISP, ALGOL, Pascal, Modula, ADA, C y Simula (Figura 1).


1960
1970
1980
1990
LISP
Algol
Simula
Smalltalk
Pascal
C
Modula
Flavors
Ada
Objective-C
LOOPS
Eiffel
C++
Object-Oriented Pascal
Actor
CLOS
LENGUAJES NO ORIENTADOS A OBJETOS
LENGUAJES HIBRIDOS ORIENTADOS A OBJETOS
LENGUAJES PUROS ORIENTADOS A OBJETOS

Simula, desarrollado por Nygaard y Dahl en la década de los 60 en Noruega, es el lenguaje que más ha influido en la POO ya que introdujo los conceptos de objeto y encapsulación, acceso externo a los atributos de los objetos, clases de los objetos, herencia y asociación dinámica.

Este lenguaje fue muy usado en Europa pero no en Estados Unidos, donde fue visto simplemente como una extensión de ALGOL específicamente orientado a simulación. En Estados Unidos el lenguaje más popular en programación orientada a objetos es Smalltalk.

El lenguaje Smalltalk, se desarrolló a principios de la década de los 70, es considerado el lenguaje de programación orientado a objetos número uno y el más puro dentro de los lenguajes orientados a objetos, ya que se diseñó especialmente para hacer programas orientados a objetos.

ADA, desarrollado en 1975 por el Departamento de Defensa de EE.UU., es un lenguaje rico en estructura modular; la modularidad la trabaja mediante funciones y procedimientos, la abstracción de datos por medio de paquetes y la concurrencia por medio de tareas.

Otros lenguajes que han sido diseñados especialmente para hacer programas orientados a objetos son Eiffel y Actor.

Eiffel, desarrollado por Bertrand Meyer, es una versión moderna de Simula. En esta versión se eliminaron algunos elementos que no son muy necesarios y el conjunto de bibliotecas se hizo más extenso.

Actor es un lenguaje de programación orientada a objetos basado en Pascal.

El lenguaje con el que se dió más auge a la programación orientada a objetos es C++ que es una extensión de C, lenguaje de programación muy popular y portable.

Objetive-C es un superconjunto de C que incorpora elementos de Smalltalk. Objetive-C soporta tipos de datos abstractos, herencia y operadores sobrecargados.

Object Pascal fue diseñado por Niklaus Wirth y un grupo de ingenieros en computación de Apple. Este es una extensión de Pascal que soporta la noción de tipos de datos abstractos, métodos y herencia.

Las extensiones orientadas a objetos de LISP son las siguientes: Flavors que es la extensión más notable de este lenguaje, LOOPS y CLOS.

A los lenguajes C++, Objective-C, Object Pascal, Flavors, LOOPS y CLOS se les llama lenguajes híbridos orientados a objetos

porque a los lenguajes de programación estructurada de los que provienen se les "pego un parche" para que fueran lenguajes de programación orientados a objetos.

1.2 MODELOS DE PROGRAMACION

Las técnicas de programación han ido evolucionando como una respuesta a las necesidades de la industria.
En esta sección se presenta la evolución que ha tenido el proceso de programación.

1.2.1 ORIENTADO A PROCEDIMIENTOS

El primer modelo de programación fue orientado a procedimientos. Este modelo de programación tiene como objetivo hacer algoritmos eficientes.

En esta etapa surgió la programación estructurada. Con la programación estructurada se introdujeron los conceptos de paso de parámetros, variables locales, procedimientos y llamada a funciones. Cuando surgió la programación estructurada eran muy pocos los lenguajes que soportaban la técnica, pero en la década de los 80 este estilo de programación era el normal para diseñar y escribir programas.

Con la programación estructurada los programas son diseñados de acuerdo a las funciones que van a ser realizadas.
Con la importancia que se dió a los datos, en la programación, surgió la modularidad y con ésta el ocultamiento de datos.

1.2.2 MODULARIDAD

La modularidad consiste en dividir el programa en varias partes; a cada una de éstas se les llama módulos. Los módulos de un programa no se hacen al azar; por el contrario, un módulo está formado por un conjunto de procedimientos relacionados y datos que son manipulados por éstos. La estructura de un módulo está formada por dos partes: la pública y la privada. En la parte pública se describen las funciones del módulo que pueden ser usadas por otros procedimientos, a ésta se le llama **interfaz**, y la parte privada es donde se implantan las funciones que fueron declaradas en la interfaz. Esta estructura permite ocultar información ya que al usuario sólo se le permite ver la interfaz.

El principio de ocultamiento de información es un componente fundamental en la metodología de programación. Provee las normas de cómo pueden ser concebidas las interfaces.

En la programación modular lo más importante son los módulos, es por esto que al diseñar un programa se debe tomar en cuenta la relación que hay entre sus funciones para agruparlas en módulos.

El objetivo de la programación modular es que cada módulo tenga una estructura simple, sea fácil de modificar, y se puedan agregar más módulos a un sistema.

Las ventajas que se obtienen al trabajar con módulos son las siguientes:

- Cuando se modifica la implantación de algún módulo no se afectan otras partes del programa, si hay bajo acoplamiento entre ellos (ver sección 2.11).
- El usuario no puede modificar la implantación ya que sólo se le permite usar los procedimientos, pero sin tener acceso a la implantación de los mismos.
- El tamaño de cada módulo no es muy grande.
- Los datos no pueden ser manipulados por cualquier función o procedimiento.

A fin de cuentas, para integrar un módulo a un sistema, basta saber qué es lo que hace y no cómo lo hace.

Aunque la programación modular tiene muchas ventajas también tiene sus limitantes. La desventaja que tiene el trabajar con módulos es que no se pueden definir nuevos tipos. Por ejemplo, si se tiene un módulo que maneja pilas de tipo entero y más tarde se necesita otra estructura como ésta pero de tipo caracter, la segunda pila no se podrá crear ya que en el módulo se definió una única pila de tipo entero.

1.2.3 TIPOS DE DATOS ABSTRACTOS

Los tipos de datos abstractos son tipos definidos por el programador que pueden ser manipulados, en forma similar a los

tipos de datos definidos por el lenguaje.

Con los tipos de datos abstractos se tiene disponible un conjunto de operaciones que pueden ser usadas para manipular datos de ese tipo.

Los módulos y los tipos de datos abstractos están relacionados, pero no son idénticos. Un modulo se puede usar como técnica de implantación de los tipos de datos abstractos.

1.2.4 PROGRAMACION ORIENTADA A OBJETOS

Después de la programación estructurada, la evolucion mas importante en el proceso de programación es la programación orientada a objetos. En contraste con los programas estructurados, en los cuales se da mas importancia a la implantación de los procedimientos, los programas orientados a objetos son diseñados de acuerdo a los datos que van a ser procesados.

La programación orientada a objetos se ha hecho muy popular desde hace pocos años.

Este tipo de programación introdujo una nueva manera de pensar, en como dividir los problemas y dar una solucion para el desarrollo de los programas.

Cuando se habla de programación orientada a objetos se dice

que usando este modelo de programacion, el desarrollo y mantenimiento de los programas se hace más fácilmente. Sin embargo, como cualquier otro modelo de programación, éste no garantiza un camino fácil hacia un *software* perfecto. Para obtener todos los beneficios que proporciona, debe ser usado adecuadamente.

La programación orientada a objetos es una tecnica de programación - un modelo en el que los programas se organizan a través de objetos y con estos se forman las clases. Estas a su vez están jerarquizadas por medio de la herencia.

Los principales elementos son: los objetos y las clases, y sus principales mecanismos son: la abstracción, la modularidad, la encapsulación y la herencia.

Otros mecanismos de menor importancia son: la asociación dinamica y estatica, la concurrencia y la persistencia.

Estos conceptos se detallarán en el próximo capítulo.

CONCEPTOS GENERALES DE LA PROGRAMACION ORIENTADA A OBJETOS

En el capítulo anterior se mencionaron los conceptos más comunes de la programación orientada a objetos, en este capítulo se definirán tales conceptos.

2.1 ABSTRACCION

La abstracción hace énfasis en los principales detalles de lo que se quiere modelar. Es decir, identifica los objetos que interesan y destaca sus características esenciales. Esto desde luego depende del observador. Por ejemplo, a un automóvil, un conductor lo ve como algo con lo que se puede desplazar de un lugar de la ciudad a otro, realizando un mínimo de esfuerzo. En cambio para un mecánico es algo que está compuesto por un motor, ruedas, carrocería, etc.

Las abstracciones tienen propiedades estáticas y dinámicas. Las propiedades estáticas son aquellas que no hacen que un objeto cambie en cuanto a su funcionamiento y las propiedades dinámicas son el cambio en el valor de las propiedades estáticas. Para ilustrar las propiedades estáticas y dinámicas se ejemplificará de la siguiente manera:

Un coche tiene motor, transmisión, tanque de gasolina, llantas, etc. Estos componentes del automóvil son sus propiedades estáticas. En particular, le podemos cambiar las llantas que trae actualmente por otras de diferente tipo, esto es una propiedad dinámica.

2.2 ENCAPSULACION

La abstracción y la encapsulacion son conceptos complementarios; la abstraccion se enfoca en la vista externa de los objetos mientras que la encapsulación impide que se vea la parte interna de ellos.

La encapsulacion oculta los detalles de un objeto que no contribuyen en sus caracteristicas esenciales.

Cuando encendemos un radio escuchamos la música, lo que hablan los locutores y los comerciales, pero se desconoce cómo el radio capta las ondas hertzianas, ésto es un ejemplo de encapsulacion.

Hasta este momento se ha visto lo que es la modularidad, ocultamiento de datos, tipos de datos abstractos, la abstracción y encapsulación. A continuación se definirán los principales elementos de la programación orientada a objetos: los objetos y las clases.

2.3 OBJETOS

En general, un objeto es una cosa, un ente, un elemento, un cuerpo, una forma.

Desde el punto de vista de los lenguajes de programación y a nivel elemental, a los objetos los podemos implantar como registros que pueden existir en la ejecución de programas en C

o Pascal. Al igual que un registro, un objeto es una estructura que ocupa espacio en memoria durante la ejecución del sistema y está compuesto de cierto número de elementos llamados campos. La figura 2.1 representa al objeto "Alumno" que contiene los datos de un alumno, como son: el nombre, el número de cuenta, el año de ingreso a la universidad y la clave de la carrera.

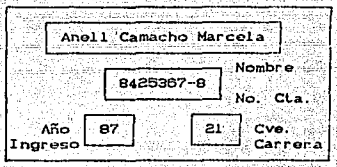


Figura 2.1 Objeto Simple

Un objeto puede contener otros objetos, tal como en Pascal un registro puede contener otro registro. La figura 2.2 representa al objeto "Calificación". El objeto Calificacion tiene los siguientes campos: calificación, periodo en que acreditó la materia, tipo de examen en que la acreditó, y también tiene el objeto "Materia". El objeto materia tiene los campos: nombre de la materia, clave, créditos y nivel.

El objeto que está dentro no puede ser compartido por otros objetos. Compartir objetos es necesario cuando varios objetos deben hacer uso de la información que contiene un mismo

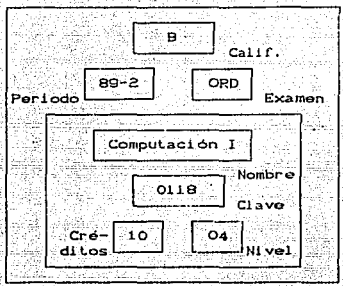


Figura 2.2 Objeto que contiene otro objeto

objeto. La razón para compartir objetos no sólo es para economizar espacio en memoria, sino también para que los cambios que se hacen en el objeto que se está compartiendo se reflejen simultáneamente en todos los objetos que están haciendo uso de él, sin tener problemas de inconsistencia.

Entre los campos que contiene un objeto también puede haber campos para referirse a otros objetos. En la figura 2.3 el objeto "Materia" está siendo compartido, ya que hay dos alumnos que cursaron la misma materia.

En la programación orientada a objetos, los objetos tienen estado, comportamiento e identidad.

El estado de un objeto lo determina un conjunto de características como son las propiedades estáticas y el valor

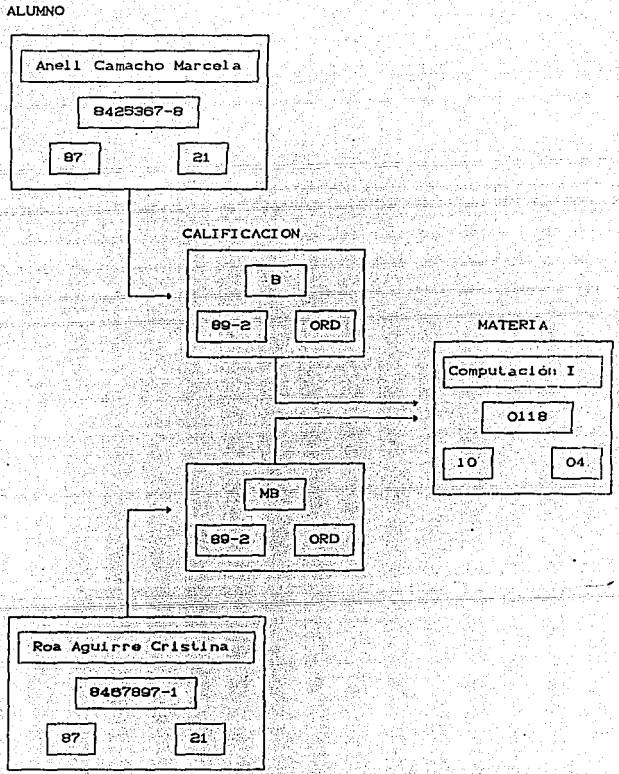


Figura 2.3 Compartiendo un objeto

de cada una de esas propiedades.

El comportamiento se refiere al funcionamiento del objeto, para qué sirve.

La identidad son las características particulares que lo distinguen de los demás objetos similares.

Para ilustrar el estado, comportamiento e identidad de un objeto se seguirá tomando el ejemplo del automóvil.

El estado de un automóvil está determinado por el motor, la transmisión, las llantas, asientos, etc., y el material con que se fabricaron; al ponerlo en marcha se puede uno desplazar por las calles en él, esto es su funcionamiento. Si se menciona que el número de placas es XYZ890 entonces, el número de placas es lo que hace que se distinga de los otros coches y por lo tanto es su identidad.

Los objetos por sí solos no existen en un sistema, éstos existen por medio de las relaciones de uso que hay entre ellos.
Los objetos sirven como operandos sobre los cuales pueden realizarse operaciones, y como operadores que realizan operaciones sobre otros.

Dependiendo de la función que realice un objeto sobre otro se

tienen las siguientes relaciones:

Actor — Es un objeto que opera sobre otros objetos, pero ningún objeto puede operar sobre él.

Servidor — Es un objeto que nunca opera sobre otros objetos; pero otros objetos si pueden operar sobre él.

Agente — Son aquellos objetos que pueden operar y ser operados por otros objetos. Esta relación es la más común.

En la figura 2.3 se presentan los objetos Alumno, Calificación y Materia. Si con esos tres objetos se quiere obtener un historial académico entonces, el objeto Materia es un servidor ya que no puede operar sobre ningún otro objeto, pero sobre él está operando el objeto Calificación. El objeto Alumno, va a operar sobre el objeto Calificación para saber que calificaciones pertenecen a un alumno determinado. Como el objeto Calificación está operando sobre Materia y está siendo operado por Alumno, es un objeto agente, y Alumno es un objeto Actor.

Al conjunto de operaciones que un objeto puede realizar sobre otro se llama protocolo.

En la programación estructurada cuando un procedimiento hace uso de otro procedimiento o función, se dice que está haciendo

una llamada a una función. En contraste, en la programación orientada a objetos, cuando un objeto llama a otro se denomina envío de mensaje.

2.4 CLASES

El concepto de clase es fundamental en la programación orientada a objetos.

Una clase es una estructura que agrupa un conjunto de objetos que tienen las mismas características en cuanto a estructura y funcionamiento.

Para no confundir los términos de objeto y clase se debe tener presente que:

- Los objetos son elementos que deben ser creados durante la ejecución de un sistema; las clases son descripciones estáticas de un conjunto de posibles objetos (las instancias de la clase). En tiempo de ejecución se tienen solamente objetos; en el programa sólo se observan clases.
- Un campo en un objeto corresponde a un atributo de la clase de la cual el objeto es una instancia.

Dado que no todos los componentes de una clase están disponibles para cualquier otra clase y para los usuarios, éstos están divididos en dos partes: la interna y la externa. En la parte externa, llamada interfaz de la clase, se declaran todas las operaciones que se aplican a las instancias de la

clase.

Las operaciones que se aplican a las instancias de la clase son llamadas métodos.

En la parte interna de una clase se implantan todas las funciones que fueron declaradas en la interfaz.

La interfaz de una clase está dividida en tres partes:

Pública Es la visible y accesible para el usuario de la clase.

Protegida Estas declaraciones son privadas en el sentido de que no pueden ser usadas por cualquier clase, sólo sus subclases (ver definición más adelante) tienen acceso a ellas.

Privada Es la representación (estructura) del objeto. Las declaraciones que se encuentran aquí sólo pueden ser usadas por los métodos de la clase.

La diferencia entre una subclase y un usuario de la clase es que el usuario de la clase crea instancias de la misma y envía mensajes a esos objetos mientras que una subclase crea nuevas clases basándose en la clase de la cual proviene.

2.5 CREACION, INICIALIZACION Y DESTRUCCION

Durante la creación se debe destinar espacio en memoria para contener un nuevo objeto y asociar ese espacio con un nombre.

Hay dos tipos de variables: las automáticas y las dinámicas. Las variables automáticas son aquellas que son creadas cuando se encuentra su declaración en un procedimiento y el espacio que ocupan es liberado cuando el procedimiento termina. Las variables dinámicas son aquellas en las que el espacio que van a ocupar en memoria debe ser apartado explícitamente por el programador.

En la inicialización se dan valores iniciales a los objetos.

Cuando un objeto ya no se usa, es destruido automáticamente o mediante funciones que proporciona el lenguaje.

Los objetos Alumno, Materia y Calificación que se vieron en el ejemplo de la sección anterior, contienen los datos de un alumno, la calificación y la materia: estos valores no deben ser destruidos porque sirven para la creación de un historial académico.

Cuando los objetos no son destruidos al terminar un programa, se dice que son persistentes.

2.6 HERENCIA SENCILLA

Las clases se relacionan de manera jerárquica, es decir, al agrupar en una clase un conjunto de objetos con características y funcionamiento común, de ellas se pueden ir derivando nuevas clases que hereden sus propiedades, pero estas nuevas clases son más específicas. Por ejemplo, si una clase estudiantes es una subclase de una clase personas y ésta es una subclase de la clase mamíferos, entonces la clase estudiantes hereda las propiedades de las clases mamíferos y personas (Figura 2.4).

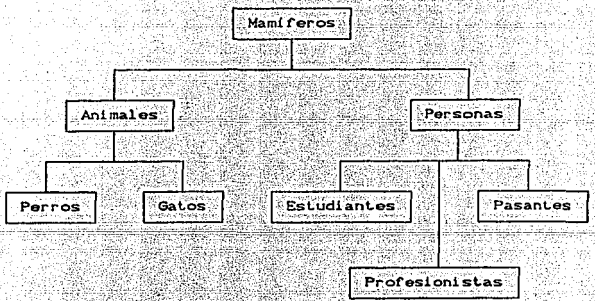


Figura 2.4 Jerarquía de clases

La clase mamíferos es llamada superclase ya que de ella se derivan nuevas clases. En general, una superclase es toda clase que hereda su funcionamiento a otra clase.

Los beneficios que se obtienen con el mecanismo de herencia son los siguientes:

- Reutilización de *software*. Se debe a que los métodos de una clase es heredado a otra clase y por lo tanto no se escribe nuevamente.
- Código compartido. El código se comparte cuando varios programadores usan la misma clase.
- Consistencia en la interfaz. Como varias subclases heredan funcionamiento y propiedades de una misma superclase, se está seguro de que el funcionamiento de las instancias de las clases herederas es similar.
- Ocultamiento de información. Cuando un programador usa un componente de *software* sólo necesita saber cómo funciona, por lo tanto sólo se le proporciona la interfaz y no los detalles de cómo fue implantado.
- Polimorfismo. Con el polimorfismo se generan algoritmos de alto nivel para luego adaptarlos a aplicaciones particulares. Este tema se verá con más detalle en otra sección.

2.7 HERENCIA MULTIPLE

En la figura 2.8 se muestra la superclase escuelas, con las subclases primaria, secundaria, preparatoria y universidad.

En la sección anterior vimos que la clase estudiante heredó las propiedades de personas y mamiferos, pero un estudiante puede ser de primaria, secundaria, preparatoria, universidad,

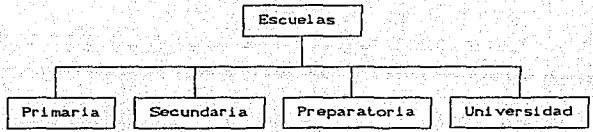
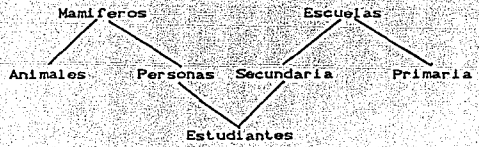


Figura 2.5 Jerarquía de Escuelas

etc. Si, en particular, está en la secundaria entonces también hereda las propiedades de la clase secundaria, a esto se le llama herencia múltiple.

En general, si una clase A hereda las propiedades de dos o más clases, de diferente árbol, se dice que hay herencia múltiple.



Usando el mecanismo de herencia se va de un nivel general a un nivel particular.

2.8 ASOCIACION ESTATICA Y DINAMICA

La asociación es simplemente el significado que se da a los atributos en alguna parte del programa, o el sentido de una construcción particular.

La asociación puede hacerse durante la compilación, donde el código del programa es encontrado por primera vez, durante el ligado, cuando los resultados de las diferentes compilaciones son combinados, y en tiempo de ejecución, cuando un programa es finalmente ejecutado.

Antes de ver que es la asociación estática y dinámica se definirá que es un identificador, un valor y un tipo. Un identificador es simplemente un nombre, este es el medio por el cual un programador denota un dato para ser manipulado. El término valor describe el contenido, en la memoria de la computadora, asociado con un identificador. Un tipo puede ser asociado, dependiendo del lenguaje, con una variable o con un valor.

En lenguajes estáticamente tipificados[1], los tipos son asociados con variables. En estos lenguajes los tipos sirven, entre otras cosas, para denotar el conjunto o rango de valores que las variables pueden contener.

Los tipos también son usados para determinar el sentido de una operación. Por ejemplo, la expresión

a + b

[1]Los lenguajes estáticamente tipificados son aquellos en los que la verificación de los tipos se hace durante la compilación.

es una suma de enteros si a y b fueron declarados de tipo entero, y reales si fueron declarados como reales.

En lenguajes dinámicamente tipificados[2], los tipos no se asocian con variables, sino con valores. En estos lenguajes de acuerdo al tipo del valor es el tipo que se le asocia a la variable.

La asociación estática significa que los tipos de todas las variables y expresiones se fijan durante la compilación, es decir, la asociación de un mensaje con un método está basada en las características estáticas de las variables. Si en un programa se declara una variable x de tipo entero y luego se encuentra la siguiente operación

```
x = x + 2
```

aquí se está haciendo una asociación estática ya que en este momento se sabe de que tipo es x.

La asociación dinámica significa que los tipos de todas las variables y expresiones son conocidas hasta la ejecución del sistema. La asociación dinámica y el polimorfismo (ver sección 2.10) están muy relacionadas ya que en las funciones polimórficas el significado que se les asocia se sabe hasta

[2]Los lenguajes dinámicamente tipificados son los lenguajes que hacen la verificación de tipos durante la ejecución de un programa.

que el sistema es ejecutado.

2.8 REFINAMIENTO Y REEMPLAZO

Como ya se dijo anteriormente, una subclase es más específica que una superclase, esto se debe a que tiene métodos y datos propios, estos métodos y datos se adicionan al protocolo de la superclase.

Algo diferente ocurre cuando una subclase define un método con el mismo nombre de un método que utiliza una superclase. el método de la subclase oculta (o invalida) al método de la superclase.

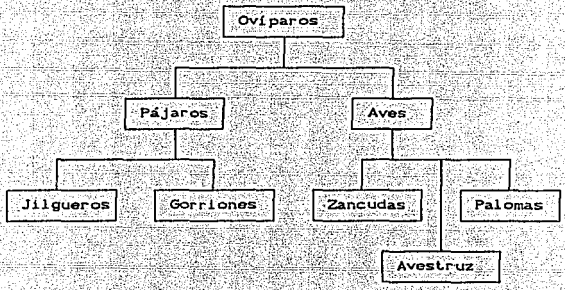
Cuando un mensaje es enviado a un objeto, el método de búsqueda empieza a localizar al método asociado con la clase del objeto. Si el método no es encontrado, entonces se busca en la superclase inmediata, y así hasta encontrarlo. Es por esta razón que el método de una superclase es invalidado porque cuando empieza a buscar la primera clase que examina es la subclase.

Los propósitos por los cuales una subclase invalida a un método de una superclase son dos: reemplazo y refinamiento.

En el método de reemplazo se sustituye totalmente al método de la superclase, es decir, el código de la superclase nunca se ejecuta cuando las instancias de la subclase son manipuladas. En el método de refinamiento se incluye, como parte de su

funcionamiento, la ejecución del método heredado de la superclase. Así, el funcionamiento de la superclase es preservado y aumentado.

A fin de ejemplificar estos conceptos, a continuación se muestra la clasificación de los ovíparos según su tamaño corporal.



Algunas de las características de los ovíparos son que vuelan y cantan, entonces dos métodos de la clase ovíparos son volar y cantar.

El avestruz no vuela, corre. En la clase de las avestruces se puede definir un método que se llame volar, pero volar va a significar que corren muy rápido; de esta manera se reemplaza el método volar de la clase ovíparos.

Los jilgueros son pájaros que tienen un canto melodioso. Si en la clase jilgueros, al método cantar de la clase oviparos se le agrega que el canto es melodioso entonces se refina el método cantar.

2.10 POLIMORFISMO

El término polimorfismo significa "muchas formas". Así por ejemplo, en biología, una especie polimórfica es una especie que se caracteriza por la ocurrencia de diferentes formas o colores en un organismo.

En lenguajes de programación, un objeto polimórfico es cualquier entidad, tal como una variable o argumento de una función, que permite valores de diferentes tipos durante la ejecución de un sistema. Las funciones polimórficas son aquellas que tienen argumentos polimórficos.

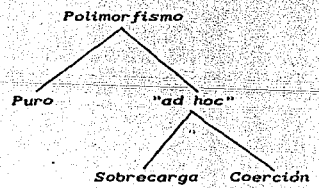
En los lenguajes de programación la forma más común de polimorfismo es la sobrecarga (*overloading*), el símbolo + es un símbolo sobrecargado ya que puede hacer la suma de reales y de enteros.

El polimorfismo puede ser visto en términos de abstracción de alto nivel y de bajo nivel.

En la abstracción de alto nivel se provee la estructura de un algoritmo pero sin los detalles esenciales, mientras que la

abstracción de bajo nivel provee los detalles necesarios para la ejecución. El poder del polimorfismo radica en que permite que algoritmos de alto nivel sean escritos una sola vez, y sean reutilizados repetidamente con abstracciones de bajo nivel. En los lenguajes orientados a objetos, el polimorfismo ocurre como resultado del mecanismo de paso de mensajes y herencia.

El polimorfismo puro ocurre cuando a una función simple se le pueden aplicar argumentos de diferentes tipos. En el polimorfismo puro se tiene una función y un número de interpretaciones diferentes. Otro tipo de polimorfismo ocurre cuando se tiene un número de funciones diferentes denotadas por el mismo nombre. Esta situación es conocida como sobrecarga o polimorfismo ad hoc. Dentro del polimorfismo ad hoc también se encuentra la coerción (ver sección 2.10.3).



2.10.1 Objetos Polimórficos

Con excepción de la sobrecarga, el polimorfismo en los lenguajes orientados a objetos es posible sólo por la

existencia de objetos polimórficos. Un objeto polimórfico es un objeto con muchas formas, es decir, es un objeto que permite valores de diferente tipo.

2.10.2 Polimorfismo puro

Muchos autores reservan el término polimorfismo para cuando una función puede ser usada con una variedad de argumentos, en oposición a la sobrecarga en la que existen múltiples funciones definidas con un solo nombre. La habilidad para hacer funciones polimórficas es una de las técnicas más poderosas de la programación orientada a objetos. Esto permite que el código sea escrito una sola vez, en una abstracción de alto nivel, y luego sea ajustada a una variedad de situaciones.

Un ejemplo de polimorfismo puro es una función que hace la suma de dos números.

Suma (a1,a2)

Esta función realiza la operación independientemente del tipo de los parámetros que le proporcionen.

2.10.3 Sobrecarga

En la sobrecarga lo polimórfico es el nombre de la función ya que se tienen diferentes cuerpos de funciones denotados por el mismo nombre. En este tipo de polimorfismo el código que se

ejecuta depende de los argumentos que se dan. El ejemplo más común de sobrecarga es el operador +. El código que es generado por el compilador cuando hace una suma de enteros es diferente al que se genera cuando hace una suma de reales. Aquí también encontramos un polimorfismo diferente, la coerción. La coerción ocurre cuando un valor de un tipo es convertido en otro de diferente tipo.

Sean

suma (real1, real2),

suma (entero1, entero2) y

suma (complejo1, complejo2)

Estas tres funciones tienen el mismo nombre; al recibir dos parámetros para realizar la suma, se verifica primero de que tipo son y luego se ejecuta la función correspondiente. En este caso hay una sobrecarga.

Para ver un ejemplo de coerción, considérese la función

suma (real1, real2)

si se quiere realizar la suma de un entero y un real, se debe convertir el parámetro de tipo entero a real.

2.11 INTERCONEXION DE LOS COMPONENTES DE *SOFTWARE*

Otra cosa importante en la programación orientada a objetos es la interconexión de los componentes de *software*.

Estas interconexiones pueden ser vistas en términos de visibilidad y dependencia.

Un término que es usado frecuentemente para describir la visiblidad es el alcance (*scope*) de los identificadores. El alcance o medio ambiente de un identificador es la parte del programa en donde este identificador puede ser utilizado (visible).

La dependencia es usada para relacionar una parte de un sistema de *software* con otro.

2.11.1 Acoplamiento

Los conceptos de acoplamiento y cohesión fueron introducidos para evaluar eficientemente la estructura de los módulos.

El acoplamiento describe la relación de uso entre módulos y la cohesión describe la relación que hay entre los elementos del módulo. Una reducción de la interconexión de modulos (o clases) se alcanza por medio de la reducción del acoplamiento. Para tener un buen diseño la cohesión de los elementos de un módulo (o clase) debe maximizarse. Esto no se hace sólo para que las clases sean más entendibles, sino también para que sean fácilmente extraídas para una aplicación particular y para que sean reutilizadas nuevamente.

Las formas de acoplamiento son las siguientes:

- Acoplamiento interno de datos,
- Acoplamiento global de datos,
- Control (o secuencia) de acoplamiento,
- Acoplamiento paramétrico,
- Acoplamiento de subclases.

El acoplamiento interno de datos ocurre cuando en un módulo se permite que otro módulo modifique los valores locales de los datos. Este tipo de acoplamiento debe evitarse cuanto sea posible.

El acoplamiento global de datos ocurre cuando dos o más módulos (clases) son ligados en una estructura de datos común.

Control o secuencia de acoplamiento es cuando un módulo debe realizar operaciones en cierto orden fijo, pero el orden es controlado por otro módulo.

El acoplamiento paramétrico ocurre cuando un módulo invoca el servicio de una rutina, pero se relaciona sólo mediante el número y el tipo de parámetros proporcionados y el tipo de valor regresado. De todas las formas de acoplamiento que se han visto hasta ahora, ésta es la mejor opción.

El acoplamiento de subclases es particular para la programación orientada a objetos. Este acoplamiento describe la relación de una subclase con una superclase (o superclases

en el caso de herencia múltiple). Por medio de la herencia una instancia de la subclase puede ser tratada como si fuera una instancia de la superclase.

Para reducir el grado de acoplamiento entre objetos, Karl Lieberherr desarrolló una herramienta llamada Ley de Demeter.

Ley de Demeter (forma débil). Dentro de un método M es posible tener acceso a los datos de los siguientes objetos:

1. Objetos que son argumentos del método M,
2. Objeto receptor del método,
3. Objetos globales,
4. Objetos temporales creadas dentro del método.

y también es posible que estos objetos puedan enviar mensajes dentro del método M.

Ley de Demeter (forma fuerte). Solamente a los siguientes objetos se les permite acceso o envío de mensajes, dentro de un método dado.

1. Objetos que son argumentos del método M,
2. Objeto definidos definidos en la clase que contiene al método,
3. Objetos globales,
4. Objetos temporales creadas dentro del método.

Las dos formas de la ley de Demeter son similares, sólo difieren en el punto número dos donde, la forma débil los

permite a las clases herederas, el acceso a los atributos de la superclase. En la forma fuerte los atributos de la superclase solamente pueden ser manipulados por los métodos de ella y para que una clase heredera pueda acceder a tales atributos se debe contar con funciones de acceso.

2.12 CONCURRENCIA

Algunos sistemas deben realizar varios procesos simultáneamente. Si se contara con un procesador que fuera capaz de realizar múltiples tareas al mismo tiempo o con varios procesadores, no habría problema pero si no se tiene ese tipo de procesador entonces, se debe asignar una cantidad de tiempo a cada proceso. Si los procesos no se ejecutan completamente en el tiempo que les fue asignado se les vuelve a dar tiempo y así hasta que se realicen completamente. El algoritmo que asigna el tiempo a cada proceso es tan eficiente que parece que se están ejecutando simultáneamente.

Cuando se tiene solamente un procesador y aparentemente se ejecutan varios procesos a la vez se dice que hay concurrencia.

CONCEPTOS DE LA PROGRAMACION ORIENTADA A OBJETOS EN C++

Hasta ahora se han visto los conceptos de la programación orientada a objetos en forma general. En este capítulo se detallará la forma en que se aplican tales conceptos en el lenguaje de programación orientado a objetos C++.

3.1 OBJETOS, CLASES Y METODOS

En C++ se tienen dos tipos de archivos: los de interfaz y los de implantación.

En este lenguaje, a los métodos de una clase se les llama funciones miembro y la implantación de las mismas se encuentra en el archivo de implantación.

La definición de una clase empieza con la palabra clave class seguida por el nombre de la clase; el cuerpo está formado por la parte privada, la protegida y la pública.
La forma general de una clase es la siguiente:

```
class nom_clase  {
private:
        ...          representación de la clase
protected:
        ...          clases derivadas
public:
        ...          mensajes
}
```

Las palabras claves private, protected y public pueden ir en cualquier orden y repetrise cuantas veces se quieran.

Anteriormente se mencionó que la implantación de las funciones miembro se encontraba en el archivo de implantación, pero también es posible encontrar la implantación de aquellas funciones cuyo cuerpo es muy pequeño, en el archivo de interfaz.

Cuando la implantación de una función miembro está en el archivo de interfaz se dice que es una función inline.

En C++ un objeto es una instancia de una clase.

En el listado 3.1 se muestra un ejemplo de un archivo de interfaz y en el listado 3.2 un ejemplo de un archivo de implantación.

Listado 3.1 En este listado se tiene la definición de la clase alumno. El estado de cada objeto de esta clase es el nombre, número de cuenta, año de ingreso y la clave de la carrera, y su comportamiento es poder asignarle un valor a cada uno de esos campos.

```
//   Alumno.h
class Alumno
{
private:
     char nombre [35];   // nombre del alumno
     int awo_ingreso;    // año en que ingreso a la facultad
     int cve_carrera;    // clave de la carrera que esta cur-
                         // sando
protected:
     char No_Cta [10];   // número de cuenta
public:
     Alumno ();          // función que inicializa los atribu-
```

```
                                    // de la clase alumnos
    // Funciones miembro de la clase Alumno
    void pide_nombre ();
    void pide_cta ();
    void pide_awo ();
    void pide_cve ();
    void trae_nombre (char nom[]);
    void trae_cta (char cta[]);
    int trae_awo ();
    int trae_cve ();
};
```

Listado 3.2 Este listado contiene la implantación de las funciones miembro de la clase Alumno.

```
//   Alumno.cpp

#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include "Alumno.h"
const int  TAMAÑO = 9;    // constantes que serán usadas
const int  LIMITE1 = 0;   // en algunas de las funciones
const int  LIMITE2 = 99;  // miembro


Alumno::Alumno()
// Función constructora.  Esta función inicializa
// los campos de las instancias de la clase Alumno
{
    int i;
    for (i=0; i < 35; nombre[i++]= '\x0');
    for (i=0; i < 10; No_cta[i++]= '\x0');
    awo_ingreso = 0;
    cve_carrera = 0;
}


void Alumno::pide_nombre ()
// Función que pide el nombre del alumno
{
    cout << "\tDame el nombre del alumno \n";
    gets (nombre);
}


void Alumno::pide_cta ()
// Función que pide el número de cuenta del alumno
// El número de cuenta debe ser de 9 caracteres
{
    int aux;
    do
```

```
    {
        cout << "\tDame el número de cuenta:\n";
        gets (No_Cta);
        aux = strlen (No_Cta);
    } while ( aux != TAMAÑO );
}
void Alumno::pide_awo ()
// Función que pide el año de ingreso a la facultad
// Esta función sólo acepta valores entre 0 y 99
// pero si se desea, pueden ser cambiados
{

    do
    {
        cout << "\tDame el año de ingreso: ";
        cin >> awo_ingreso;
    }while((awo_ingreso<=LIMITE1) ||(awo_ingreso>LIMITE2))
}


void Alumno::trae_nombre (char nom[])
// Esta función regresa una copia de lo que está almacenado
// en nombre
{
    strcpy (nom, nombre);
}


void Alumno::trae_cta (char cta[])
// Función que regresa una copia del valor que hay en
// No_Cta
{
    strcpy (cta, No_Cta);
}


int Alumno::trae_awo ()
// Función que regresa una copia del valor que hay en
// awo_ingreso
{
    return (awo_ingreso);
}


int Alumno::trae_cve ()
// Esta función regresa una copia del valor que
// está almacenado en cve_carrera
{
    return (cve_carrera);
}
```

Las clases son el medio por el cual C++ logra la abstracción y

encapsulación de los datos.

3.2 MENSAJES

El envío de mensajes en C++ es conocido como invocación a una función miembro.

Las funciones miembro se pueden invocar directa e indirectamente.

Para invocar a una función miembro directamente primero se pone el nombre del receptor (nombre de la instancia), seguido por un punto (.), luego el nombre de la función miembro y una lista de parámetros entre parentesis. Si la función miembro no tiene parámetros los parentesis van vacios, y esto se hace para diferenciarla de la invocación a un dato.

El acceso indirecto a una función miembro se hace usando el símbolo de indirección ->.

El listado 3.3 muestra como se pueden accesar las funciones miembro en forma directa e indirecta. El programa cuenta el número de líneas que tiene un texto que se introdujo por medio del teclado; para esto se implantó la clase contador.

Listado 3.3 Demostración del acceso indirecto y directo a una función miembro.

```
//  Cont.h
//  interfaz de la clase Contador
```

```
class Contador
{
private:
     unsigned int valor;
public:
     Contador ();                    // Función constructora
     void incrementa ();
     void decrementa ();
     unsigned int accesa_valor ();
};


//   Cont.cpp
//      Implantación de las funciones miembro de la clase
//   Contador


#include "Cont.h"

Contador::Contador ()
//función constructora
{
     valor = 0;
}


void Contador::incrementa ()
// función que incrementa el contador
{
     if (valor < 65535)
          valor++;
}


void Contador::decrementa ()
// Función que decrementa el contador
{
     if (valor > 0)
          valor--;
}


unsigned int Contador::accesa_valor ()
// Función que regresa una copia del dato almacenado
// en valor
{
     return (valor);
}


//   Contlin.cpp
//   Programa que cuenta el número de líneas leídas,
//   desde el telado
```

```
#include <iostream.h>
#include <stdio.h>
#include "Cont.h"

main ()
{
     char c;                  // var. a la que se le asignan
                              // los caracteres leidos del
                              // teclado
     Contador nl, *anl;       // Declaracion de una instancia
                              // de Contador y un apuntador a
                              // él.
     cout<<"Introduce un texto \n";
     // El acceso que se hace, en las siguientes lineas,
     // a las funciones miembro, es en forma directa
     while ((c=getchar ()) != EOF)
          if (c == "\n")
               nl.incrementa ();
     cout<<"Numero de lineas leidas: "<<nl.accesa_valor ();
     while (nl.accesa_valor () != 0 )
          nl.decrementa ();
     anl = &nl;               // asigna a anl la dirección de
                              // nl
     // acceso indirecto usando apuntador
     cout<<"Introduce un texto \n";
     while (( c = getchar ()) != EOF)
          if ( c == "\n" )
               anl->incrementa ();
     cout<<"Numero de lineas leidas: "<<anl->accesa_valor ();
     return (0);
}
```

3.3 CREACION, INICIALIZACION Y DESTRUCCION

La inicialización implícita es facilitada en C++ por el uso de la función constructora. Una función constructora es un método que tiene el mismo nombre de la clase. La forma de invocar a una función constructora es la siguiente

```
nom_fun_constructora  nom_objeto;
```

Cuando en alguna parte del programa se hace lo anterior, se

crea un objeto del tipo de la función constructora que se invocó.

En el archivo contlin.cpp, del listado anterior se tiene la siguiente declaración:

```
Contador n1;
```

en ese momento se crea un objeto de tipo Contador, y también se inicializa.

En C++ también se puede definir una función que se invoca automáticamente cuando el espacio en memoria para un objeto es liberado. Esta función es llamada destructora.

El uso más común de la función destructora es liberar cualquier almacenamiento que fuera asignado por la función constructora. Todas las funciones destructoras tienen el mismo nombre que la clase con la que están asociadas, excepto que el nombre está precedido por el símbolo ~. La función destructora se llama cuando un objeto de la clase deja de existir. Esto ocurre cuando termina su ámbito o cuando termina el programa.

En el ejemplo que se usa para ilustrar el mecanismo de herencia sencilla (siguiente sección), se tiene una función constructora y una destructora.

3.4 HERENCIA SENCILLA

En este lenguaje a las superclases se les llama clases bases y a las subclases se les llama clases derivadas.

Mediante el mecanismo de herencia una clase base hereda sus atributos a una clase derivada y así se crea una jerarquía en forma de árbol.

La forma general para heredar una clase a otra es:

```
class nom_c_derivada: acceso nom_c_base
{
    .....
};
```

donde el acceso puede ser público (public) o privado (private). Por omisión el acceso es privado.

Si el acceso es público, todos los elementos públicos de la clase base son heredados como elementos públicos a la clase derivada, y los elementos protegidos de la clase base también son heredados como protegidos a la clase derivada.

Si el acceso es privado, entonces todos los elementos públicos y protegidos de la clase base se convierten en elementos privados de la clase derivada, es decir, los elementos públicos y protegidos de la clase base no van a poder ser usados por las clases herederas de la clase derivada.

En el listado 3.4 se tiene el programa Posfija.cpp. Este programa convierte una expresión en notación infija (los operandos son letras) a notación posfija; para convertir una expresión infija a posfija se debe tomar en cuenta la precedencia de los operadores y también los paréntesis. Para resolver el problema se usó la estructura de datos Pila.

La clase Pila tiene las funciones mete, saca, vacia, tope e imprime. La implantación de estas funciones se hizo mediante la clase Lista, así, esta última heredó su funcionamiento a la clase Pila.

Listado 3.4 Demostración del uso de herencia.

```
//    Lista.h
//    Esta clase va a ser usada para implantar la estructura
//    de datos Pila.

const int Max_elem = 30;.  // Número máximo de elementos que
                           // puede almacenar

class Lista
{
     char *Lista;     // Lista guarda la dirección de un
                      // apuntador a caracteres
     int nmax;        // dimensión del arreglo
     int nelem;       // número de elementos que hay en la lista
public:
     Lista (int n=Max_elem)          // Función constructora
          {Lista = new char[n]; nmax = n; nelem = 0;};
     ~Lista () {delete Lista};       // Función destructora
     int pon_elem (char elem, int pos);
     int trae_elem (char& elem, int pos);
     int trae_n () {return nelem}; // regresa el número de
                                   // elementos en la lista
     void inc_n () {if (nelem < nmax) ++nelem;};
     int n_max () {return nmax;};   // regresa la dimiensión
                                    // del arreglo
     void imprime ();
};
```

```
//   Lista.cpp

#include <iostream.h>
#include "Lista.h"


int Lista::pon_elem(char elem, int pos)
// Esta función almacena a elem en la  posición pos de la
// Lista.  El usuario de la lista es quien debe incremen-
// tar la posición
{
     if (0 <= pos && pos < nmax)
     // Si la posición está dentro del rango
     {
       lista[pos] = elem;
      return 0;
     }
     else
      return (-1);
}


int Lista::trae_elem (charb elem, int pos)

// Esta función, dada la posición en la Lista, regresa una
// copia del elemento que se encuentra en dicha posición
// almacenándolo en la variable elem.
{
     if (0 <= pos && pos < nmax)
     {
        elem = lista[pos];
        return (0);
     }
     else
      return (-1);
}


void Lista::imprime()

// Esta función imprime los elementos de la lista en la
// consola
{
     for (int i = 0; i < nelem; ++i)
         cout << lista[i] <<"\n";
}


//   Pilas.h
//   clase en C++ que implanta la estructura de datos pila

#include "lista.h"

class Pila : private Lista   // La clase pila hereda los
```

```
                                // atributos de la clase Lista
{
    int tope;
public:
    Pila () {tope = 0;};     // funciones constructoras
    Pila (int n) : Lista(n) {tope = 0;};
    int mete (char elem);
    int saca (char& elem);
    // La función vacia regresa un 1 si la pila está vacía
    // y 0 en otro caso
    int vacia () {if(tope == 0) return 1; else return 0;};
    char Tope ();
    void imprime ();
};


// pila.cpp

#include <iostream.h>
#include "pila.h"


int Pila::mete (char elem)
// Esta función mete a elem en el tope de la Pila e
// incrementa dicho tope
{
    int m = n_max ();   // Obtiene el tamaño de la pila
    if (tope < m)       // Si no se ha llenado la pila
    {
       pon_elem (elem,tope++);
       return (0);
    }
    else
       return (-1);
}


int Pila::saca (char& elem)
// Esta función saca de la pila el elemento que  se  encuentra
// en el tope, almacenándolo en la variable elem  y decrementa
// el tope de la Pila
{
    if ( !vacia ())     // checa que la pila no esté vacía
    {
       trae_elem (elem, --tope);
                        // Obtiene el elemento en el tope
                        // de la pila y tope es el lugar
                        // que está disponible para alma-
                        // cenar otro elemento
       return (0);
    }
    else
       return (-1);
}
```

```
void Pila::imprime ()

// Función que imprime los elementos de la pila en pantalla
// En este caso no se uso la función imprime de la clase
// Lista porque una pila y una lista son estructuras que
// acceden de formas diferentes a sus elementos.
{
     char elem;
     for (int i = tope-1; i >= 0; --i)
     {
        trae_elem (elem,i);
        cout << elem << "\n";
     }
}


char Pila::Tope ()
// Esta función regresa el último elemento que se ha
// almacenado en la Pila
{
     char elem;
     int   Top;

     Top = tope - 1;  // top es el índice del último
                      // elemento que se almacenó
     trae_elem (elem,Top);
     return (elem);
}


//   Posfija.cpp
//   Programa que convierte una expresion en notación infija a
//   notación posfija

#include <iostream.h>
#include <ctype.h>
#include <conio.h>
#include "Pila.h"
const int FALSE = 0;
const int TRUE = 1;
void inicia (char *in);
int operador (char c);
int operando (char &c);
int correcta (char *in);
void expresion (char *in);
void inf_pos (Pila p, char *in, char *pos);

main ()
{
     Pila p(30);          // se crea una instancia de la clase
                          // Pila
     char infija [30], posfija[30];
                          // arreglos para almacenar las ex-
```

```
                        // presiones
    char aux;
    do
    {
       expresion ( infija );
       inf_pos (p,infija,posfija);
       cout <<"\n\tLa expresión infija es: "<<infija<<"\n\t";
       cout <<"La expresión en forma posfija es: "<<posfija;
       do
       {
          cout<<"Deseas convertir otra expresión? (s/n): ";
          aux=getche();
       }while ((toupper(aux) != 'S') && (toupper(aux) != 'N
    } while (toupper (aux) != 'N');
}


void inicia (char *in)
// Función que inicializa una cadena con ceros
{
    while (*in)
        *in++='\x0';
}


int operador (char c)
// función que recibe un caracter y regresa un 1 si es
// operador y 0 en caso contrario.
{
    switch (c)
    {
          case '+':
          case '-':
          case '*':
          case '/':
                return (TRUE);
          default:
                return (FALSE);
    }
}


int operando (char &c)
// Esta función verifica si el caracter que recibe es letra;
// la letra que recibe la convierte a mayúscula
{
    c=toupper(c);
    return ((c>='A') && (c<='Z'));
}


int correcta (char *in)
// Función que verifica que la expresión infija sea correcta.
// Una expresión se considera incorrecta si pasa alguno de los
```

```
// siguientes casos:
//      - el número de paréntesis que abren es diferente al
//        número de paréntesis que cierran
//      - el primer caracter es operador
//      - algún caracter no es operador ni operando
//      - antes de un paréntesis que abre hay un operando
//        o después un operador
//      - después de un paréntesis que cierra hay un operando
//        o antes, un operador
{
    int pa=0, pc=0;     // paréntesis que abren y cierran
    char *aux1, *aux2; // variables auxiliares
    aux1=in;
    if (operador (*in))
        return (FALSE);
    else while (*aux1)
        {
            if (*aux1 =='(')
                pa++;
            else if (*aux1 == ')')
                pc++;
            else if (!operador(*aux1) && (!operando (*aux1)))
                return (FALSE);
            aux1++;
        }
    if (pa != pc)
       return (FALSE);
    else
        {
          aux1=aux2=in;
          while (*aux2)
          {
            aux2++;
            if (*aux1=='(' && operador (*aux2))
                  return (FALSE);
            else if ( operador (*aux1) && *aux2==')')
                  return (FALSE);
            else if (*aux1==')' && operando (*aux2))
                  return (FALSE);
            else if (operando (*aux1) && (*aux2=='('))
                  return (FALSE);
            else if (operador (*aux1) && operador (*aux2))
                  return (FALSE);
            else if (operando (*aux1) && operando (*aux2))
                  return (FALSE);
            aux1++;
          }
        }
    return (TRUE);
}


void expresion (char *in)
// Esta función pide una expresión en notación infija
```

```
// mientras ésta no sea correcta
{
     do
     {
          inicia (in);
          cout<<'\n\tDame la expresión en notación infija: \n";
          cin>>in;
     } while ( !correcta (in) );
}


void inf_pos (Pila p, char *in, char *pos)
// Esta función convierte una expresión en notación infija a
// una en notación posfija, y para ello usa una instancia  de
// la clase pila en la cual se guardan los operadores.
{
     char elem;
     inicia (pos);
     while (*in)
     {
          if (operando (*in))
             *pos++ = *in++;
          else if (*in=='(')
               p.mete (*in++);
          else if ( *in == ')')
               {
                    if ( p.Tope () =='(')
                    {
                         p.saca (elem);
                         in++;
                    }
                    else while (p.Tope() != '(')
                         {
                              p.saca (elem);
                              *pos++ = elem;
                         }
               }
          else if ((*in == '/')  || (*in == '*'))
               {
               if ((p.Tope()=='*')  || (p.Tope()=='/'))
               {
                    p.saca (elem);
                    *pos++ = elem;
                    p.mete (*in++);
               }
               else
                    p.mete (*in++);
               }
          else if ((*in == '+')  || (*in == '-'))
               {
               if ((p.Tope () == '(')  || (p.vacia () ))
                    p.mete (*in++);
               else
                 {
```

```
                    p.saca (elem);
                    *pos++ = elem;
                    p.mete (*in++);
                  }
            }
      }
      while ( !p.vacia())
      {
          p.saca (elem);
          *pos++=elem;
      }
}
```

3.5 HERENCIA MULTIPLE

En la sección anterior se observó que una clase puede heredar el funcionamiento de otra clase y a ésto se le llama herencia sencilla. C++ también soporta la herencia múltiple, es decir, una clase derivada puede heredar simultáneamente de más de una clase base.

La forma general es la siguiente:

```
class nom_c_derivada: acceso nom_c_base1,
                      acceso nom_c_base2, ...
{
    ....
};
```

En la sección 2.3 se usaron los objetos Alumno, Calificación y Materia para mostrar las relaciones que hay entre objetos. En esta sección se hizo la implantación de la clase Alumno, Materias y Calificación, para mostrar el mecanismo de herencia

múltiple en C++.

La clase Calificación contiene un campo para la calificación, el semestre y el tipo de examen en que acreditó la materia; además de estos campos debe contener una referencia para saber a qué alumno y a qué materia corresponde la calificación. Una forma que permite que la clase Calificación tenga referencia a la clase Alumno y Materia es multiheredando los atributos de estas dos clases. En el listado 3.5 se tiene la implantación de la clase Materia y Calificación, la implantación de la Clase Alumno está en el listado 3.2.

Listado 3.5 Demostración del uso de herencia múltiple

```
//    Materias.h
//    Implantación de la clase Materias
class materias
{
private:
     char nombre[30];          // nombre de la materia
     int  creditos;            // créditos de la materia
     int  nivel;               // semestre al que pertenece
protected:
     int  clave;               // clave de la materia
public:
     Materias ();              // función constructora
     // funciones miembro
     void pide_nommat ();
     void pide_clave ();
     void pide_creditos ();
     void pide_nivel ();
     void trae_nommat (char mat[]);
     int  trae_clave ();
     int  trae_creditos ();
     int  trae_nivel ();
};


//    Materias.cpp

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

```
const int PRIMERO = 1     // variables para indicar el primer
const int ULTIMO = 10;    // y último semestre de una carrera


Materias::Materias ()
// Funcion constructora
{
    int i;
    for (i=0; i<30; nombre[i++]='\x0');
    clave = 0;
    creditos = 0;
    nivel = 0;
}


void Materias::pide_nommat ()
// Función que lee el nombre de la materia
{
    cout<<"Dame el nombre de la materia: \n";
    cin>>nombre;
}


void Materias::pide_clave ()
// Funcion que lee la clave de la materia
// La clave de una materia está entre 1 y 9999
{
    do
    {
        cout<<"Dame la clave de la materia: ";
        cin>>clave;
    } while ((clave < 1) || (clave > 9999));
}


void Materias::pide_creditos ()
// Función que lee los créditos de la materia
// Una materia tiene como mínimo 6 créditos y como
// máximo 18
{
    do
    {
        cout<<" Dame el numero de creditos: ";
        cin>>creditos;
    } while ( (creditos < 6) || (creditos >18 ));
}


void Materias::pide_nivel ()
// Función que lee el nivel al que corresponde la materia
{
    do
    {
        cout<<"Dame el nivel: ";
```

```
        cin>>nivel;
    } while ( (nivel < PRIMERO ) || (nivel > ULTIMO ));
}


void Materias::trae_nommat (char mat[])
// Función que regresa una copia del valor del dato almacenado
// en nombre
{
    strcpy(mat,nombre);
}


int Materias::trae_clave ()
// Función que regresa una copia del valor almacenado en
// clave
{
    return (clave);
}


int Materias::trae_creditos ()
// trae_creditos regresa una copia del valor almacenado en
// creditos
{
    return (creditos);
}


int Materias::trae_nivel ()
// Función que regresa una copia del valor del dato almacenado
// en nivel
{
    return (nivel);
}


//   Implantación de la clase Calificación
//   Calif.h

#include "alumno.h"
#include "Materias.h"

//   Esta clase está multiheredando de Alumno y Materia
//   El listado de Alumno se encuentra al inicio del capítulo

class Calificacion: public Alumno, public Materias
{
private:
    char calific[3];          // variable que almacena la cali-
                              // ficación obtenida
    char tipo_examen[4];      // variable para almacenar el tipo
                              // de examen en que la acreditó
    char periodo[5];          // variable para almacenar el
```

```
                                    // semestre en que la acreditó
public:
     Calificacion ();
     void pide_calif ();
     void pide_examen ();
     void pide_periodo ();
     void trae_calif (char calif[]);
     void trae_examen (char tex[]);
     void trae_periodo (char per[]);
};


Calificacion::Calificacion()
// Función constructora
{
     int i;
     for (i=0; i<3; calific[i++]='\x0');
     for (i=0; i<4; tipo_examen[i++]='\x0');
     for (i=0; i<5; periodo[i++]='\x0');
}


void Calificacion::pide_calif()
// Función para leer la calificación obtenida
{
     cout<<" Qué calificación obtuvo: ";
     cin>>calific;
}


void Calificacion::pide_examen ()
// Lee el tipo de examen en que acreditó la materia
{
     cout<<"Tipo de examen en que la acredito: ";
     cin>>tipo_examen;
}


void Calificacion::pide_periodo ()
// Función que lee el semestre en que la materia fue
// acreditada
{
     cout<<" En que periodo la acreditó: ";
     cin>>periodo;
}


void Calificacion::trae_calif (char calif[])
// Regresa una copia del valor almacenado en calific
{
     strcpy (calif,calific);
}
```

```
void Calificacion::trae_examen (char tex[])
// Regresa una copia del valor almacenado en tipo_examen
{
     strcpy (tex, tipo_examen);
}


void Calificacion::trae_periodo ( char per[])
// regresa copia del valor almacenado en periodo
{
     strcpy (per, periodo);
}



//    Prueba.cpp
//    Este programa crea un archivo de objetos de tipo
//    calificación

#include <iostream.h>
#include <fstream.h>
#include "Calif.h"
#define INDICE 15


main ()
{
     int i;
     ofstream archcal ("Cal.dat");  // abrir archivo para
                                    // escritura
     Calificacion calif[INDICE];    // arreglo de objetos
     for ( i=0; i<INDICE; i++)
     {
        calif[i].pide_cta ();       // multiheredó de Alumnos
        calif[i].pide_clave ();     // y Materias
        calif[i].pide_calif ();
        calif[i].pide_examen ();
        calif[i].pide_periodo ();
     }
     // Se escriben en el archivo los datos que se
     // obtuvieron en el ciclo anterior
     archal.write ((char*)calif, INDICE*sizeof(Calificacion));
     archcal.close();
     return (0);
}
```

En el ejemplo anterior la clase Calificación multiheredó de las clases Alumno y Materia.

Estas tres clases podrían ser usadas para generar un historial

académico mediante archivos ya que en la clase Alumno se tienen los datos de un alumno, en la clase Materia se tienen los datos de una materia y como calificación multiheredó de estas dos entonces, el archivo de calificaciones tendría una calificación, tipo de exámen en que se aprobó, el periodo, el número de cuenta del alumno y la clave de la materia a la que pertenece esa calificación.

3.6 POLIMORFISMO

En C++ se tiene polimorfismo en tiempo de compilación y en tiempo de ejecución.

3.6.1 Polimorfismo en tiempo de compilación

El polimorfismo en tiempo de compilación se logra a través de la sobrecarga de funciones y de operadores.

Al escribir un programa que calcule las raíces de una ecuación de segundo grado ($ax^2 + bx + c$), puede suceder que el usuario dé el coeficiente del término cuadrático igual a cero, en este caso se tendría que enviar un mensaje al usuario, indicándole que el coeficiente no es válido; también podría suceder que, además de dar el coeficiente del término cuadrático igual a cero, diera el valor del coeficiente del término lineal igual a cero; en este caso no es ecuación y también se le tendría que indicar al usuario que esos valores no son válidos. Para evitar esto, en el programa raices.cpp, la función que calcula las raíces tiene como parametros de

salida la solución de la ecuación y un parámetro que indica si las raíces son: reales, imaginarias, la ecuación es de primer grado o no fue ecuación.

Para mostrar el resultado obtenido, se sobrecargo una función que imprime: 1) Un mensaje indicando que la función no era ecuación, 2) Un número real si la función era de primer grado, 3) Las raíces reales de la ecuación, o 4) Las raíces imaginarias. Dependiendo de los parámetros que le sean pasados, el compilador ejecuta la función adecuada.

Listado 3.6 Sobrecarga de funciones.

```
//   Imprimir.h

class Imprimir
{
public:
     // Función sobrecargada
     void Imprime ( float r1);
     void Imprime ();
     void Imprime ( float real, float imag1, float imag2 );
     void Imprime ( float r1, float r2);
};

//   imprimir.cpp

#include <iostream.h>
#include "Imprimir.h"


void Imprimir::Imprime ( float r1 )
// Imprime la solución de una ecuación de primer grado
{
     cout<<"La ecuación es de primer grado y su raiz es "<<r1;
}


void Imprimir::Imprime ()
// Solo escribe un mensaje para indicar que no se tuvo
// ecuación
{
```

```
    cout<<"No es una ecuación";
}
void Imprimir::Imprime ( float real, float imag1, float imag2 )
// Imprime las raices complejas de una ecuación de segundo
// grado
{
    cout<<"Las raices son: "<<r<<"+i"<<r1<<" y "<<r<<"-i"<<r2;
}


void Imprimir::Imprime (float r1, float r2 )
// Imprime las raices reales de una ecuación de segundo grado
{
    cout<<"Las raices son: "<<r1<<" y "<<r2;
}


//    Raices.cpp
//    Este programa calcula las raices de una ecuación de
//    segundo grado y para mostrar el resultado hace uso
//    de la función sobrecargada Imprime.

#include <iostream.h>
#include "Imprimir.h"
#include <math.h>

void raices (float, float, float, float*, float*, float*, int*);
enum {primer, no_ecuacion, no_reales, reales};

Imprimir    print;      // declaración de una instancia de
                        // Imprimir
main()
{
    int codigo;
    float a, b, c,      // Coeficientes de la ecuación
          real, raiz1,
          raiz2;        // Raices
    cout<<"Dar los coeficientes de la ecuación:\n";
    cin>>a>>b>>c;
    raices (a, b, c, &real, &raiz1, &raiz2, &codigo);
    switch (codigo)
    {
        case primer:        print.Imprime (raiz1);
                            break;
        case no_ecuacion:   print.Imprime ();
                            break;
        case no_reales:     print.Imprime (real, raiz1, raiz2);
                            break;
        default:            print.Imprime (raiz1,raiz2);
                            break;
    }
}

// Esta función calcula las raices de una ecuación de segundo
```

```
// grado con coeficientes reales a, b, c y cod indica que tipo
// de raices o de ecuación se tuvo
void raices (float a, float b, float c, float *r,
             float *r1, float *r2, int *cod)
{

     double disc;
     if (!a && b)
     {
          *r1=-c/b;
          *cod=primero;
     }
     else if ( !a && !b)
          *cod=no_ecuacion;
     else if ((disc = b*b - 4*a*c ) < 0 )
          {
               *r= -b/(2*a);
               *r1=*r2= sqrt (-disc)/(2*a);
               *cod= no_reales;
          }
     else
        {
               *r1= ( -b + sqrt (disc))/(2*a);
               *r2= ( -b - sqrt (disc))/(2*a);
               *cod= reales;
        }
}
```

Las funciones constructoras también pueden sobrecargarse.

Los operadores solamente pueden usarse para realizar operaciones sobre los tipos definidos por el lenguaje (enteros, reales, etc.) y no sobre tipos definidos por el usuario.

En C++ se pueden sobrecargar los operadores para que puedan ser usados en tipos definidos por el usuario. Aquí, es importante mencionar que en C++, cada vez que una función miembro es invocada, automáticamente se pasa un apuntador al objeto que la invocó, y es posible conocer ese apuntador con

la palabra this.

En el listado 3.7 se sobrecargo el operador + para sumar vectores y el operador = para asignar un vector a otro.

Listado 3.7 Operadores sobrecargados

```
//    vector.h

#include <iostream.h>

class Vector
{
private:
      float x;                        // entrada (1,1) del vector
      float y;                        // entrada (1,2) del vector
public:
      Vector () { x=0.0; y=0.0; }; // Función constructora
      void asigna (float cx, float cy)
                  { x=cx; y=cy; }; // Asigna los valores
                                      // leidos, al vector
      Vector operator=(Vector vect);
      Vector operator+(Vector vect);
      void despliega (void)           // muestra en la pantalla
                  { cout<<x<<","<<y<<"\n"; // el vector
};


//    vector.cpp

#include "vector.h"


Vector Vector::operator=(Vector vect)
// sobrecarga del operador =, para asignar un vector
// a otro
{
      x=vect.x;
      y=vect.y;
      return *this;
}


Vector Vector::operator+(Vector vect)
// Sobrecarga del operador +, para sumar dos vectores
{
      Vector aux;
      aux.x = x+vect.x;
      aux.y = y+vect.y;
      return (aux);
}
```

```
// pvector.cpp

#include "vector.h"

main()
{
    float a;
    Vector V1, V2, V3; // creación e inicialización de tres
                       // vectores
    V1.asigna (1,1);
    V2.asigna (2,2);
    V3 = V1 + V2;      // aplicación de la sobrecarga
    V1.despliega();
    V2.despliega();
    V3.despliega();
}
```

3.6.2 Polimorfismo en tiempo de ejecución

El polimorfismo en tiempo de ejecución se logra por medio de las funciones virtuales. Las funciones virtuales son las funciones miembro de una clase, que son declaradas usando el calificador virtual y pueden redefinirse en una o más clases derivadas.

Hay dos tipos de funciones virtuales.

El primer tipo de funciones virtuales son aquellas que son declaradas como virtuales en la clase base de la siguiente forma:

```
virtual tipo nom_función (lista de parámetros);
```

Cuando se declaran estas funciones en la clase derivada, puede

prescindirse de la palabra virtual, pero los parámetros y el tipo deben ser los mismos.

La implantación de estas funciones se encuentra en la clase base y la clase derivada pude tener su propia versión.

El otro tipo de funciones virtuales son las funciones virtuales puras y la forma de declararlas es la siguiente:

```
virtual tipo nom_fun (lista de parámetros) = 0;
```

Con este tipo de funciones se forza a las clases derivadas a que tengan su propia versión ya que de no tenerla se provoca un error.

Las funciones virtuales pueden invocarse como cualquier otra función miembro pero sólo con el uso de apuntadores es como se logra el polimorfismo en tiempo de ejecución.

Los polígonos son figuras con muchos lados. Entre las cosas que interesa saber sobre un polígono están: el perímetro y el área; pero la forma de calcular el área y el perímetro de un triángulo es diferente a la de un rectángulo. Con el uso de las funciones virtuales es posible declarar las funciones área y perímetro en una clase base y redefinirlas en varias clases derivadas.

En el listado 3.8 se definieron las funciones virtuales pide_lados, area y perimetro, en la clase Poligono, y luego se redefinieron en la clase Triangulo.

Listado 3.8 Demostración del uso de funciones virtuales

```
//    poligono.h

class Poligonos
{
protected:
      float lado1;        // un polígono tiene, al menos tres
      float lado2;        // lados.
      float lado3;
public:
      Poligonos ();
      // Funciones virtuales
       virtual void pide_lados () = 0;
       virtual float perimetro () = 0;
       virtual float area() = 0;
};


//    Poligono.cpp

#include "Poligono.h"
Poligonos::Poligonos()
{
      lado1=0.0;
      lado2=0.0;
      lado3=0.0;
}


//    Triang.h
//    Esta clase recibe como herencia los atributos de polígono

#include "poligono.h"


class Triangulos: public Poligonos
{
public:
      Triangulos ();
      // Declaración de las funciones virtuales que
      // recibió  de Poligonos
      void pide_lados ();
      float perimetro();
      float area();
};
```

```
//  Triangulo.cpp

#include <iostream.h>
#include <math.h>
#include "triang.h"

Triangulos::Triangulos ()
// Función constructora
{
     lado1=lado2=lado3=0.0;
}


void Triangulos::pide_lados ()
// Función con la que se obtienen los lados de un triángulo
{
     cout<<"\tDame la longitud de los lados del triángulo:\n";
     cin>>lado1>>lado2>>lado3;
}


float Triangulos::perimetro ()
// Función que calcula el perímetro de un triángulo
{
     float per;
     per=lado1+lado2+lado3;
     return (per);
}


float Triangulos::area()
// Función que calcula el área del triángulo
{
     float A;
     double S;
     // Si los lados son iguales se trata de un triángulo
     // equilátero
     if ((lado1==lado2) && (lado2==lado3))
     {
          A=((lado1*lado2)/4.0)*sqrt(3.0);
          return (A);
     }
     // Si no es triángulo equilátero el cálculo del área se
     // hace con la siguiente fórmula:
     // A = (s*(s-lado1)*(s-lado2)*(s-lado3))^1/2
     // donde s = ( lado1 + lado2 + lado3 ) / 2
     else
     {
          S=(lado1+lado2+lado3)/2.0;
          if ( (S <= lado1 || S <= lado2) || S <= lado3)
               return (0);
          else
          {
               A=sqrt(S*(S-lado1)*(S-lado2)*(S-lado3));
```

```
                return (A);
            }
        }
}


//  prueba.cpp
//  prueba del polimorfismo en tiempo de ejecución

#include <iostream.h>
#include "triang.h"

main ()
{
    Poligonos *pol;             // declaración de un apuntador de
                                // la clase base
    Triangulos triangulo;       // declaración de una instancia
                                // de la clase Triangulos
    pol=&triangulo;             // polimorfismo en tiempo de
    pol->pide_lados();          // ejecución
    cout<<"El perímetro del triángulo es: "<<pol->perimetro();
    cout<<"El area es: "<<pol->area();
    if ( pol->area () == 0)
        cout <<" Error en los datos ";
    return (0);
}
```

En el capítulo anterior se mencionó el reemplazo de métodos. Con el uso de las funciones virtuales C++ logra el reemplazo de un método ya que una clase derivada puede tener su propia versión de una función con el mismo nombre del método de una clase base.

En C++ la asociación estática es lograda en tiempo de compilación mediante la llamada a funciones estándares, llamada a funciones sobrecargadas y operadores sobrecargados, y la asociación dinámica se logra mediante el uso de funciones virtuales.

3.7 VISIBILIDAD

En C++ el acceso a la información está controlado mediante el uso de las palabras clave public, private y protected.

La forma débil de la ley de Demeter puede cumplirse declarando todos los atributos de la clase como protegidos. La forma fuerte se cumple cuando los atributos son declarados como privados.

En secciones anteriores se dijo que a los elementos privados sólo tienen acceso las funciones miembro de la clase, pero C++ cuenta con una clase de funciones que sin ser miembros de la clase pueden manipular a los datos privados y protegidos de las instancias de dichas clase. Estas funciones son llamadas funciones amigas y se declaran usando el calificador friend en la declaración de la función en la clase.
La declaración de una función amiga es la siguiente:

```
tipo friend nombre (Lista de parámetros);
```

donde los parámetros son de tipo clase.

En el siguiente listado se tiene la implantación de la clase Persona. La clase persona se usa para pedir los datos de un individuo y para mostrar sus datos en pantalla, se uso una función amiga.

Listado 3.9 Demostración del uso de funciones amigas

```
//   Persona.h

class Persona
{
private:
     char nombre [35];          // nombre de la persona
     int edad;                  // edad
     char sexo [20];            // sexo
     char edo_civil [11];       // estado civil
public:
     Persona ();
     void pon_nombre ();
     void pon_edad ();
     void pon_sexo ();
     void pon_edo_c ();
     // Función amiga
     void friend escribe (Persona p);
};
//   Persona.cpp

#include <iostream.h>
#include <stdio.h>
#include "Persona.h"

Persona::Persona ()
// Función constructora
{
     int i;
     for( i=0; i<35; nombre[i++]='\x0');
     for( i=0; i<2; sexo[i++]='\x0');
     for( i=0; i<11; edo_civil[i++]='\x0');
     edad=0;
}


void Persona::pon_nombre()
// Lee el nombre de la persona
{
     cout<<"Dame el nombre: ";
     gets (nombre);
}


void Persona::pon_edad ()
// Lee la edad de la persona
{
     cout<<"Dame la edad: ";
     cin>>edad;
}


void Persona::pon_sexo ()
```

```
// Lee el sexo de la persona
{
     cout<<"Sexo: ";
     cin>>sexo;
}


void Persona::pon_edo_c ()
// Lee el estado civil de una persona
{
     cout<<"Estado civil (soltero, casado, divorciado): ";
     cin>>edo_civil;
}


void escribe(Persona p)
// Función amiga de la clase Persona
// Esta función tiene acceso directo a los elementos privados
// de las instancias de la clase Persona
{
     cout<<"NOMBRE\t"<<p.nombre<<"\n";
     cout<<"EDAD\t"<<p.edad<<"\n";
     cout<<"SEXO\t"<<p.sexo<<"\n";
     cout<<"ESTADO CIVIL\t"<<p.edo_civil<<"\n";
}


//   Datosper.cpp

#include<iostream.h>
#include"Persona.h"

main()
{
     Persona p;          // Declaración de uno objeto
                         // de tipo persona
     p.pon_nombre();
     p.pon_edad();
     p.pon_sexo();
     p.pon_edo_c();
     escribe (p);        // Llamada a una función amiga
     return(0);
}
```

Programar con funciones amigas no es esencial en C++, pero en algunas ocasiones sirven como enlace entre dos o más clases y también para sobrecargar operadores.

CONCLUSIONES

Los lenguajes y técnicas de programación han evolucionado como una respuesta a las necesidades y requerimientos de la industria. En el proceso de evolución de las técnicas de programación las dos más importantes son: la programación estructurada, en la cual los programas son diseñados en relación con las funciones que van a ser realizadas, y la programación orientada a objetos, en la que los programas se diseñan tomando en cuenta la estrecha relación que hay entre los datos y las funciones que van a manipularlos.

En la actualidad, se habla mucho de la POO y se podría pensar que este modelo de programación es una moda, pero en realidad ya tiene muchos años que se empezó a gestar y hasta hace pocos tomo forma.

La POO es una realidad, un estilo de programación que llego para quedarse pero para obtener todos los beneficios que promete debe usarse correctamente y también se debe contar con un LPOO.

La importancia que ha adquirido la programación orientada a objetos se debe en gran parte a que proporciona mecanismos para la abstracción y encapsulación de datos, para el polimorfismo y la herencia de atributos.

El mecanismo de herencia es muy importante ya que a partir de la definición de clases ya existentes, se pueden crear nuevas clases con características específicas y con ésto se logra algo tan importante como es la reutilización del *software*.

En C++, las clases son el mecanismo mediante el cual logra desarrollar los conceptos de abstracción y encapsulación de datos; proporciona herramientas para soportar el mecanismo de herencia, que es tan importante en la programación orientada a objetos, y mediante el uso de la sobrecarga de funciones, operadores y funciones virtuales se logra otro de los conceptos importantes de este estilo de programación: el polimorfismo.

BIBLIOGRAFIA

1. Aguado, Joaquín y González, Luis A., Introducción a C++, México, Notas de la Maestría en Ciencias de la Computación, UACP y P del CCH, UNAM, 1993, 72 pp.

2. Booch, Grady, Object Oriented Whit Applications, EE.UU., Ed. The Benjamin/Cummings Publishing Company Inc., 1991, 540 pp.

3. Budd, Timothy, An Introduction to Object-Oriented Programming, EE.UU., Ed Addison-Wesley, 1991, 399pp.

4. Durham, Tony, "To Program is to Understand", Sign Publications, Special Silver Anniversary Supplement, EE.UU., 1992.

5. Gibson, Elizabeth, "Objects-Born and Bred", BYTE, EE.UU., october, 1990.

6. Goldshlager, Les y Lister, Andrew, Introducción Moderna a la Ciencia de la Computación. Con un enfoque algorítmico, México, Ed. Prentice-Hall Hispanoamericana, 1986, 322pp.

7. Goldstein, Charles M., "Object Oriented Programming", The Journal, EE.UU., 1990.

8. Holzner, Steve, C++ Programming, EE.UU., Ed Brady Publishing, 1991, 551pp.

9. Kelly, Al y Pohl, Ira, Lenguaje C. Introducción a la Programación, EE.UU., Ed Addison-Wesley Iberoamericana, 1987, 392pp.

10. Kerr, Ron, "Object Strike Silver", Sign Publications, Special Silver Anniversary Supplement, EE.UU., 1992.

11. Key, Stewart M, "The Natural History of objects", Sign Publications, Special Silver Anniversary Supplement, EE.UU., 1992.

12. Khoshafian, Serag y Abnous, Razmik. Object Orientation. Concepts, Languajes, Databases, User Interfaces. EE.UU., Ed Wiley, 1990. 434 pp.

13. Ladd, Robert. "Super Paradigm". Computer Languajes. EE.UU., February, 1989.

14. López Gaona, Amparo. Notas del Curso Introducción al Lenguaje C. Publicaciones de Matemáticas, Comunicación interna No. 171. México, 1990.

15. Meyer, Bertrand. "Eiffel the legacy to Simula". Sign Publications. Special Silver Anniversary Supplement. EE.UU., 1992.

16. Meyer, Bertrand. Object Oriented Software Construction. Great Britain. Ed Prentice Hall, 1988. 534 pp.

17. Schildt, Herbert. Using Turbo C++. EE.UU., Ed. Borland Osborne/ Mc Graw Hill, 1990.

18. Sethi, Ravi. Lenguajes de Programación. Conceptos y Constructores. EE.UU., Ed. Addison-Wesley Iberoamericana, 1992.

19. Stroustrup, Bjarne. "Tracing the roots of C++". Sign Publications. Special Silver Anniversary Supplement. EE.UU., 1992.

20. Stroustrup, Bjarne. "What is Object-Oriented Programming". IEEE. EE.UU., May, 1988.

21. Wegner, Peter. "Concepts and Paradigms of Object Oriented Programming". ACM PRESS. EE.UU., Junio, 1990.

22. Wegner, Peter. "Learning the language". BYTE. EE.UU., March, 1989.

23. Winblad, Ann; Edwards, Samuel; King, David. Object-Oriented Software. EE.UU., Ed. Addison-Wesley. 1990.