Capítulo

2

PROGRAMACIÓN C++

2.1 Definiciones básicas en Orientación a Objetos

Antes de iniciar la Programación Orientada a Objetos es conveniente conocer los conceptos teóricos que este paradigma involucra. En esta sección se presentan tales conceptos. Con el propósito de ilustrar algunos de ellos se presentan ejemplos con código C++, aún cuando la sintaxis de C++ se explicará con todo detalle en las secciones posteriores.

Los conceptos de la orientación a objetos surgen de la programación orientada a objetos, sin embargo, estos conceptos se han extendido a otros ámbitos de la informática, como son: las bases de datos y el análisis y diseño de sistemas, interfaces orientadas a íconos, sistemas operativos, etc. No obstante, aún muchos términos son usados para el mismo concepto y muchas veces el mismo término es usado con diferente significado. Es decir, no existe un estándar definido. Es por esto que en esta sección se presentan los principales conceptos de la Orientación a Objetos, así como algunos de sus sinónimos.

Objeto: se han dado gran cantidad de definiciones de objeto, se presentan algunas de éstas.

Booch da las siguientes [Booch]:

"Un objeto es una entidad tangible que exhibe algunas conductas bien definidas"

"Un objeto tiene estado, conducta e identidad; la estructura y la conducta de objetos similares se definen en clases comunes; los términos instancia y objeto son intercambiables"

Peter Wegner define un objeto como [Wegner]:

"Un objeto enmarca el estado de la computación en forma encapsulada. Cada objeto tiene una interfaz de operaciones que controla el acceso al estado encapsulado. Las operaciones determinan la conducta del objeto"

Alan Snyder define un objeto como sigue [Snyder]:

"Un objeto es una entidad identificable que juega un rol visible de prov

eer servicios a un cliente que los puede solicitar"

Lécluse, Richard y Velez en su artículo "un modelo de datos orientado a objetos", presentan una definición formal de objeto, esta es [Lécluse]:

"Un objeto $\mathcal{O}$ es un triplete (i,v,m) donde i es un identificador, v es un conjunto de valores (que puede ser vacío) y m es un conjunto de métodos (que puede ser vacío)."

En el artículo [Lécluse] se formaliza también el concepto de *identificador, valor* y *método* que aparecen en esta definición.

Analizando todas estas definiciones se puede decir que los objetos son *"entidades"*; no son *simplemente datos*, ni *simplemente acciones*.

Son entidades que muestran algún comportamiento reflejado por sus operaciones que actúan sobre su estado, o sobre sus datos.

Algunos autores usan *instancia, instancia de clase* o *entidad,* para referirse a un objeto. En todas las definiciones anteriores se menciona que un objeto tiene un *estado,* una *conducta*, tiene *operaciones* o *servicios* y que de alguna manera están ubicados dentro de *clases.* En los siguientes párrafos se aclaran estos conceptos.

Estado de un objeto: Booch define el estado de un objeto de la siguiente manera [Booch]:

> *"El estado de un objeto abarca todas las propiedades (usualmente estáticas) del objetos más los valores*
>
> *(usualmente dinámicos) de cada una de estas propiedades"*

Alan Snyder define el concepto de *variable de estado* para referirse al estado de un objeto [Snyder]

> *"Una variable de estado sirve como una realización concreta de los datos asociados con un objeto. Una variable de estado puede ser asociada con un único objeto o con múltiples objetos. Las variables de estado son usualmente definidas en la implementación del objeto junto con los métodos que pueden leer y escribir estas variables de estado"*

Se puede decir entonces, en términos muy orientados a la programación, que el estado de un objeto lo determina el valor (en un tiempo dado) de las variables o datos del objeto.

Conducta de un objeto: Booch define la conducta de un objeto como sigue [Booch]:

"La conducta es cómo un objeto actúa y reacciona, en términos de los cambios de su estado y el paso de mensajes"

Se puede decir que la conducta de un objeto está determinada por sus *métodos* (funciones miembro, operaciones o servicios, según otros autores); pero qué es un método?, en este sentido Alan Snyder lo define como [Snyder]:

"Un método es un procedimiento que ejecuta los servicios. Típicamente un objeto tiene un método para cada operación que soporta. Los métodos son frecuentemente definidos en la implementación del objeto permitiendo que las variables de estado sean leídas y escritas"

Clase: los objetos similares son agrupados en clases, las cuales reúnen los atributos y operaciones comunes a todas sus instancias. Peter Wegner define una clase en los siguientes términos [Wegner]:

"Una clase especifica la conducta de una colección de objetos con operaciones comunes"

Por su parte Booch define una clase de la siguiente manera [Booch]:

"Una clase es un conjunto de objetos que comparten una estructura común y una conducta común"

Algunos autores usan el término *tipo* para referirse a una clase.

Una clase puede ser el conjunto de todas las esferas en el espacio, un objeto o instancia de esta clase es una esfera particular, es decir con un radio y centro dados. En C++ la clase esfera se puede declarar como sigue:

```
class Esfera {
        float r;            //  dato o atributo para el radio de la esfera
        floar x, y, z;      //  dato o atributo para el centro de la esfera
```

```
    public:
        Esfera(float coorx. float coory, float coorz, float radio); // constructor
        ~Esfera();                  // destructor
        float volumen();            // método para calcular el volumen
        float area_superficial();  // método para calcular el área superficial
};
```

La clase Esfera tiene **miembros** que son datos (miembros dato) y miembros que son funciones o métodos (miembros método). El valor de los datos miembro en un momento dado determinan el estado del objeto y los métodos determinan la conducta.

En el siguiente fragmento de programa C++ se declara una variable de "tipo" Esfera, es decir se declara un objeto Esfera.

```
void main(void) {
        Esfera E1(2,4,5,10);
        .......
}
```

Por lo tanto, en este caso, E1 es objeto Esfera con centro (2,4,5) y radio 10.

Herencia: la herencia es uno de los mecanismos fundamentales en los lenguajes de programación orientados a objetos, además, es el principal mecanismo mediante el cual se permite la reutilización de código, algunos autores incluso dicen que un lenguaje que no soporte la herencia no es orientado a objetos, sino que *se basa en objetos*. El concepto de herencia ha sido definido también por muchos autores, Peter Wegner la define como [Wegner]:

> *"La herencia es un mecanismo para compartir el código o la conducta comunes de una colección de clases. Las propiedades compartidas se ubican en superclases y éstas se reutilizan en la definición de subclases. Los programado-*

res pueden especificar cambios incrementales en la conducta de las subclases sin modificar la clase ya especificada. Las subclases heredan el código de la superclase y ellas pueden agregar nuevas operaciones y nuevas variables de instancia"

Por su parte Booch da la siguiente definición [Booch]:

"La herencia es una clasificación u ordenamiento de la abstracción"

En este sentido puede decirse que la herencia también es un importante mecanismo para diseñar un sistema de información, pues, permite ordenar y clasificar las entidades presentes en el dominio del problema.

Alan Snyder [Snyder] agrega una idea importante al concepto de herencia, y es que el objeto de la subclase no solo hereda los métodos y datos de la superclase, sino que también pueden ser redefinidos, para adecuarlos a otros requerimientos. Snyder define herencia de la siguiente forma:

"La implementación de la herencia es un mecanismo para crear la implementación de objetos en forma incremental; la implementación de un objeto se define en términos de la implementación de otro objeto. La nueva implementación puede ser extendida agregando datos a la nueva representación del objeto, agregando nuevas operaciones, y cambiando o extendiendo la definición de operaciones ya existentes"

Para el término herencia otros autores utilizan *jerarquía de clases, jerarquía de tipos,* o *jerarquía de interfaces*; para el término superclase también se utilizan los términos *clase base* o *super tipo* y para el término subclase se utiliza comúnmente *clase derivada* o *subtipo.*

La herencia se divide en *herencia simple* y *herencia múltiple*, se dice que la herencia es simple cuando una clase "hereda" de una clase

única, y que es múltiple cuando una clase hereda de varias (más de una) clases. Para más detalles puede consultarse [Wegner] o [Booch].

Por ejemplo, se puede tener una clase Superficie de la cual hereden clases tales como Esfera, Cilindro, Elipsoides. La clase Superficie tendrá todos los atributos (estados) y métodos (conductas) comunes a las clases derivadas Esfera, Cilindro y Elipsoide. Así la definición de esta clase en C++ es la siguiente:

```
class Superficie {
protected:
        x, y, z;   // coordenadas de la superficie
public:
        Superficie(float coorx. float coory, float coorz); // constructor
        ~Superficie();
        void modificaX(float nuevoX);
        void modificaY(float nuevoY);
        void modificaZ(float nuevoZ);
        float retornaX();
        float retornaY();
        float retornaZ();
}


class Esfera : public Superficie  {          // hereda de la clase Superficie
        float r;             //  dato o atributo para el radio de la esfera
                             // los atributos x, y, z no son necesarios pues los
                             // hereda de la clase Superficie
public:
        Esfera(float coorx. float coory, float coorz, float radio); // constructor
        ~Esfera();                  // destructor
        float volumen();            // método para calcular el volumen
        float area_superficial();  // método para calcular el área superficial
        void modificaR(float nuevoR);
        float retornaR();           // También hereda los métodos para                    // modificar
};
        ................
```

De manera similar se pueden definir las clases Cilindro y Elipsoide, es decir como clases derivadas de la clase Superficie.

Encapsulación: la encapsulación es uno de los conceptos fundamentales de la Orientación a Objetos, en la que es un mecanismo que permite a los programadores utilizar clases sin conocer los detalles de implementación de éstas, permitiendo que futuras mejoras o cambios en la clase no impliquen cambios en los demás módulos que utilizan tal instancia de esta clase. Además, es el concepto que marca una diferencia sustancial entre un *Tipo de Dato Abstracto* (TDA) y un objeto, pues si bien en un TDA el programador debe manipular los datos vía las operaciones, siempre tiene la libertad de alterar directamente los datos. Mientras que en una clase mediante la *encapsulación* se obliga al programador a utilizar los métodos para leer o modificar los datos o variables de la instancia de la clase (objeto). Booch define la encapsulación como sigue [Booch]:

> *"La encapsulación es un proceso mediante el cual se ocultan todos los detalles de un objeto que no contribuyen a sus características esenciales"*

Por su parte Alan Snyder la define como [Snyder]:

> *"Un objeto está encapsulado cuando el cliente puede acceder a éste solamente mediante sus pedidos"*

Hay tres conceptos muy ligados a la encapsulación, estos son *la abstracción, la interfaz de un objeto,* y *la implementación de un objeto.* La *interfaz* de un objeto es la parte visible de un objeto, es decir, *"la interfaz describe la forma en que un objeto puede ser usado"* [Snyder]; entonces la interfaz de un objeto es el conjunto de métodos y/o datos que están disponibles o accesibles en el objeto, generalmente son métodos y no datos. La *implementación* de un objeto es *"la descripción de cómo se ejecuta el conjunto de servicios asociados con el objeto"*, es decir, es cómo están implementadas las conductas del objeto.

Algunos términos utilizados por otros autores son, *ocultamiento de información o de datos* en lugar de encapsulación y *protocolo* en lugar de interfaz.

En C++ la parte pública o interfaz de una clase está conformada por todos los atributos y métodos que están después de la palabra reservada public:, esto se explica con mucho mayor detalle en las secciones siguientes.

Abstracción: el concepto de abstracción surge con los Tipos de Datos Abstractos, la Orientación a Objetos toma este concepto como una característica esencial de un objeto, Booch la define así [Booch]:

> *"La abstracción denota las características esenciales de un objeto que lo distinguen de todos los otros tipos de objetos y provee una clara definición de las fronteras conceptuales, relativas a las perspectivas del usuario".*

El concepto de abstracción está íntimamente ligado a la encapsulación, pues es justamente la encapsulación la que permite que un objeto tenga identidad propia, y sea manipulado y conocido en el sistema por sus conductas (métodos) y no necesariamente por sus datos.

Modularidad: la modularidad es un concepto que tampoco es nuevo en la Orientación a Objetos y se utiliza ampliamente en lenguajes como Modula-2. Un módulo agrupa un conjunto de procedimientos relacionados, así como los datos que ellos manipulan. Este es un principio importante que permite en buena medida la reutilización del código, además, la modularidad pretende dividir un programa o sistema en subsistemas o subprogramas lo cual contribuye ampliamente a disminuir la complejidad del software. Booch define modularidad como sigue [Booch]:

> *"La modularidad es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos acoplados y cohesivos"*

En C++ usualmente se ubica en un módulo una jerarquía de clases, es decir, una clase y todas sus clases derivadas.

Ligamento tardío y temprano: los lenguajes de programación estructurados parten del hecho incuestionable del ligamento (*binding*) temprano, es decir, que el tipo de los parámetros se determina *a priori*, en términos de lenguajes de programación, en tiempo de compilación. Este enfoque, aún cuando simplifica los lenguajes, limita mucho el poder de modelación de la realidad, es por esto que la Orientación a Objetos incluye la posibilidad del ligamento tardío, y lenguajes como Smalltalk y C++ ya la han implementado. Snyder define como sigue el ligamento temprano y tardío [Snyder]:

> *"El ligamento temprano ocurre cuando la selección del código por ejecutar ante un pedido o solicitud se hace antes de que el servicio sea pedido"*

Algunos autores prefieren usar los términos *ligamento estático* y *ligamento dinámico* en lugar de ligamento temprano y ligamento tardío respectivamente.

Concurrencia y persistencia: la *"concurrencia permite que objetos diferentes actúen al mismo tiempo"* [Booch], este concepto aún no se ha incorporado, al menos en forma explícita en los lenguajes de programación, sin embargo, existen actualmente muchos sistemas de información que funcionan concurrentemente.

Una propiedad que también poseen muchos sistemas es la persistencia, es decir, que muchas de las características del sistema se mantienen a través del tiempo y el espacio, Booch define persistencia así [Booch]:

> *"Persistencia es la propiedad mediante la cual la existencia de un objeto trasciende el tiempo (i.e. el objeto continúa existiendo después de que su creador deja de existir) y/o espacio (i.e. el objeto se mueve del lugar en el cual fue creado)."*

Polimorfismo: el concepto de polimorfismo está íntimamente ligado al concepto de ligamento tardío. En orientación a objetos tiene que ver el hecho de que existen funciones (funciones virtuales) que pueden tener diferente implementación en clases distintas, lo que permite que en tiempo de ejecución el sistema determine cuál utilizar, de acuerdo con el tipo de parámetro con que es invocada. Alan Snyder define polimorfismo en término de funciones genéricas de la siguiente forma [Snyder]:

> *"Una operación genérica tiene diferente implementación para diferentes objetos, con conductas observables diferentes".*

Algunos autores se refieren al polimorfismo usando términos como *función virtual, función genérica, función sobrecargada* o simplemente *método.*

Otro aspecto central, y sobre el cual tampoco existe consenso, es cuáles son las características que determinan qué es Orientado a Objetos y qué no lo es.

Dave Thomas en su artículo "What is an Object?" [Thomas] define cuatro conceptos fundamentales en la Orientación a Objetos; estos son:

1. Encapsulación.
2. Paso de mensajes (solicitud de operaciones genéricas).
3. Herencia de clases, y
4. Ligamento tardío y temprano.

Mientras que para Peter Wegner es suficiente tener:

1. Objetos.
2. Clases, y
3. Herencia de clases.

Esto se expresa en la siguiente cita [Wegner]

"Lenguajes que soportan solamente objetos se dicen basados en objetos, mientras que lenguajes que adicionalmente soportan clases y herencia se denominan orientados a objetos".

Pero para Booch, según lo expresa en [Booch] existen cuatro conceptos fundamentales en la Orientación a Objetos, los cual son:

1. Abstracción
2. Encapsulación.
3. Modularidad y
4. Herencia.

El lenguaje C++ es Orientado a Objetos partiendo de cualquiera de las definiciones anteriores, pues C++ soporta las clases, los objetos, las herencias simple y múltiple, la abstracción, la encapsulación, la modularidad, el ligamiento dinámico y el paso de mensajes, como se verá en las siguientes secciones.

2.2 La noción de clase en C++

C++ fue diseñado con los siguientes propósitos:

1. Mejorar al lenguaje C.
2. Soportar los Tipos de Datos Abstractos (TDA).
3. Soportar la Programación Orientada a Objetos (POO)

Gran cantidad de programadores en todo el mundo usan el lenguaje C como su principal lenguaje para la implantación de sistemas, esto debido a que:

1. *C es flexible*: Es decir es aplicable a gran cantidad de aplicaciones.

2. *C es eficiente*: Como la semántica de C es relativamente de bajo nivel, le permite al programador utilizar eficientemente los recursos de hardware en sus programas.

3. C está disponible tanto para micros como para computadoras grandes, las bibliotecas de C también están disponibles en la mayoría de los ambientes.

En los últimos años la Programación Orientada a Objetos ha tomado gran auge, por lo que cada vez más programadores adoptan este paradigma, pero cuál es el lenguaje más adecuado? Tomando en cuenta que **C++,** salvo pequeños cambios, contiene al lenguaje C, existen gran cantidad de aplicaciones desarrolladas en C. C++ es un lenguaje que soporta todas las ideas de la Orientación a Objetos y se pueden encontrar compiladores de C++ en gran cantidad de plataformas.

C++ de AT&T Bell Laboratories versión 2.0 fue diseñado por ***Bjarne Stroustrup*** como un sucesor de C. **C**++ le agrega características a C dirigidas a la programación orientada a objetos (POO), es por esto que **C**++ fue originalmente llamado "C con clases".

Borland C++ es una versión del lenguaje **C**++, implementado por la empresa Borland que tiene todas las características del **C**++ de AT&T Bell Laboratories. Este libro se refiere fundamentalmente a Borland C++, aunque la mayoría de los ejemplos aquí presentados podrían ser ejecutados en cualquier otro compilador de C++.

Algunas de las características más importantes de la programación orientada a objetos son las siguientes (éstas se trataron con detalle en la sección anterior):

Combina la estructura de datos con las funciones (métodos) dedicados a manipular estos datos. La encapsulación es lograda mediante nuevas estructuras y mecanismos para tipos de datos, *las clases*.

Permite construir nuevas clases que heredan los datos y funciones de otras clases previamente definidas (clase base), a las cuales se les pueden definir nuevos datos y acciones, o bien se pueden redefiner los datos y métodos de la clase base. Esto crea una jerarquía de clases.

El polimorfismo permite dar a una acción, un nombre o símbolo, el cual es implementado en forma diferente y apropiada en cada nivel o clase de la jerarquía.

Las principales ventajas de la POO son:

Genera programas más estructurados y modulares.
Da la capacidad de crear nuevos tipos de datos con sus operadores especializados.
Facilita la reutilización de código.

En **POO** Programación Orientada a Objetos, los datos y los procedimientos se combinan en ***Clases***, es decir, se pueden expresar mediante la siguiente relación:

Datos + Código ≅ Clase

Una clase contiene características de una entidad (sus datos) y su comportamiento (sus procedimientos y funciones llamados métodos). Por ejemplo un aeroplano se puede describir en términos físicos (número de pasajeros, empuje que genera, coeficiente de resistencia) o en términos funcionales (despega, asciende, desciende). Sin embargo, ni la descripción física ni la funcional por separado captan la esencia de un aeroplano.

En programación C tradicional, las características físicas se definen como sigue:

```
struct aeroplano  {
        int velocidad;
        int altitud;
        int flaps;
}
```

y el comportamiento del aeroplano se define por separado:

```
void  acelera()  {
.....
}
void  desacelera()  {
.....
}
```

```
      void  sube_flaps()  {
.....
}
void  baja_flaps() {
.....
}
```

En POO las características (datos o atributos) y el comportamiento (procedimientos o métodos) se combinan en una única entidad llamada ***Clase***. La ***definición*** de un aeroplano en forma de clase es la siguiente:

```
class aeroplano  {
        int  velocidad ;   // Atributos
        int  altitud;
        int  Flaps;
public:
        aeroplano();       // Constructor
        ~aeroplano();     // Destructor
        void  acelera (); // Métodos
        void  asciende();
        void  desciende();
        void  sube_flaps();
        void  baja_flaps();
};
```

En POO los procedimientos y funciones se declaran dentro de la clase y se conocen como ***métodos***. Su código se escribe como sigue:

```
aeroplano::aeroplano() {
  flaps = 0;
  velocidad= 0;
  altitud = 0;
}

aeroplano::~aeroplano() {
  ......
 }

  .....................
```

```
void aeroplano::sube_flaps() {
  flaps = 1;
}

void aeroplano::baja_flaps() {
  flaps = 0;
}
```

Una vez definida la clase, se pueden declarar objetos, es decir variables de este "tipo" usando el nombre de la clase, tal como sigue:

```
aeroplano A;
```

En el programa se pueden escribir sentencias como éstas:

```
void main() {
        aeroplano A;     // Declara un objeto de tipo aeroplano

        A.sube_flaps(); // Se le envía un mensaje al objeto A
        A.acelera();
        ......
}
```

En C++ se pueden definir clases mediante las palabras reservadas struct, union o class. Sin embargo la palabra más utilizada es class. La sintaxis general es la siguiente:

```
class nombre_de_la_clase {
        datos y funciones privados
public:
        datos y funciones públicas
} lista_de_clases;
```

Por ejemplo, considere la siguiente definición de una clase para una lista enlazada:

```
class Lista    {
  nodo *p_nodo;
```

```
   public:
  Lista();    // Constructor
  ~Lista();   // Destructor
  void inserta(int ,char*);
  void imprime();
};
```

Como la definición de una clase es a la vez la definición de un nuevo tipo se pueden declarar variables de este nuevo "tipo" llamadas **instancias**, por ejemplo, se podrían tener la siguientes declaraciones:

```
int      valor;
Lista   CLista;          // CLista es una instacia de la clase Lista
Lista   Vector[10];      // Declara un arreglo de Listas
Lista  *P_Lista;         // Declara un puntero a Lista
```

Como se nota la declaración de instancias de clases es exactamente igual que la declaración de tipos en C, por ejemplo se declara de la misma forma la variable valor de tipo entero que CLista de "tipo" Lista.

Como se mencionó anteriormente se pueden declarar clases mediante la palabra reservada struct, pues C++ amplía las capacidades que esta palabra ya poseía en C. La diferencia central está en el acceso a los miembros de la clase (ya sean estos atributos o métodos).

Así que si una clase se define con la palabra class entonces todos sus miembros por defecto son *privados*, es decir, solo accesibles por sus propios métodos. Mientras que si una clase se define con la palabra struct entonces todos sus miembros serán por defecto *públicos*, es decir sus miembros serán accesibles libremente por cualquier expresión o función fuera de su ámbito de alcance.

Sin embargo en C++ se puede controlar el acceso a los miembros de sus clases con las palabras reservadas public:, private: o protected: de acuerdo con las siguientes reglas:

private: Los miembros seguidos de esta palabra solo pueden ser accesados por las funciones miembro de la clase, es decir, declaradas en la misma clase.

protected: Los miembros seguidos de esta palabra solo pueden ser accesados por las funciones miembro de la clase o por funciones miembro de clases derivadas, este término se explica con más detalle adelante.

public: Los miembros seguidos de esta palabra pueden ser accesados en cualquier parte del programa, dentro del ámbito de validez donde fue declarada una instancia (objeto) de la clase.

Con éstas palabras reservadas C++ facilita conceptos centrales en la Orientación a Objetos, como son el *Ocultamiento de Datos* y la *Encapsulación*.

C++ posee dos funciones miembro con características muy especiales, éstas funciones son el constructor (que podrían ser varios) y el destructor (que podrían ser varios).

El *constructor* especifica cómo se reserva memoria y cómo se inicializa para un nuevo objeto de la clase, es decir, el constructor sustituye las funciones típicas de inicialización de C. Los constructores tienen una característica muy importante, y es que *son invocados automáticamente* cuando el objeto es declarado, es decir, cuando se declara una variable de un "tipo" definido por alguna clase. Esta característica es la que permite que las clases definidas por el programador se comporten de manera muy similar a los tipos predefinidos en el lenguaje.

C++ posee también un tipo especial de función miembro llamado el *destructor*, que permite destruir un objeto previamente creado con el constructor, liberando la memoria. Al igual que los constructores, *los destructores son invocados automáticamente* cuando el ámbito de validez del objeto termina, pero también pueden ser llamados explícitamente usando el operador delete de C++.

En C++ el constructor debe llevar el mismo nombre de la clase, mientras que el destructor también debe llevar el mismo nombre que la clase pero precedido del símbolo ~.

Por ejemplo:

```
class Lista   {
  nodo *p_nodo;
public:
  Lista();   // Constructor
  ~Lista();  // Destructor
  void inserta(int ,char*);
  void imprime();
};
```

Los constructores y destructores, como funciones C++ que son, pueden tener parámetros, por ejemplo, una lista podría construirse y destruirse recibiendo un parámetro que define el tamaño de la lista:

```
class Lista   {
  nodo *p_nodo;
public:
  Lista(int tamaño);    // Constructor con parámetro
  ~Lista(int tamaño);   // Destructor con parámetro
  Lista();              // Constructor
  ~Lista();             // Destructor
  void inserta(int ,char*);
  void imprime();
};
```

En el ejemplo 2.1 se presenta un ejemplo completo de la definición e implementación de la clase Persona. Para ejecutar este ejemplo abra el proyecto E21.IDE.

🖫 Ejemplo 2.1. Clase de objetos Persona.

```
// Persona.hpp
class Persona {
  char Nombre[25];
```

```
        int Edad;
public:
   Persona(char Nom[25], int Ed);
   void Muestra();
   void CambiaNombre(char NuevoNom[25]);
   void CambiaEdad(int NuevaEdad);
   void ObtieneNombre(char Nom[25]);
   int    ObtieneEdad();
};
```

// Persona.cpp

```
#include "persona.hpp"
#include <string.h>
#include <stdio.h>

Persona::Persona(char Nom[25], int Ed) {
  strcpy(Nombre,Nom);
  Edad = Ed;
}

void Persona::Muestra() {
  printf("Persona: %s\n",Nombre);
  printf("Edad: %d\n",Edad);
}

void Persona::CambiaNombre(char NuevoNom[25]) {
  strcpy(Nombre,NuevoNom);
}

void Persona::CambiaEdad(int NuevaEdad) {
  Edad = NuevaEdad;
}
void Persona::ObtieneNombre(char Nom[25]) {
  strcpy(Nom,Nombre);
}

int Persona::ObtieneEdad() {
  return Edad;
}
```

// E21.cpp

```
      #include "persona.hpp"
#include <conio.h>

// Programa principal
void main() {
  Persona P1("Carlos",20);
  Persona P2("Maria",30);
  clrscr();
  P1.Muestra();   // Se envía el "mensaje" Muestra al objeto P1
  P2.Muestra();
  getch();
  P1.CambiaNombre("Luis");
  P1.CambiaEdad(40);
  P1.Muestra();
  getch();
}
```

La salida de este programa se presenta en la Figura 2.1.

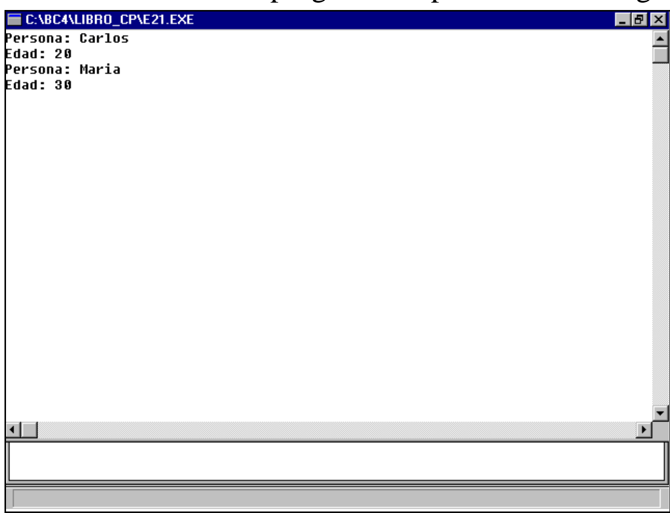


FIGURA 2.1. Salida de la Clase de objetos Persona

En el ejemplo 2.2 se ilustra la noción de encapsulación en C++, para esto se implementa una pila de números enteros mediante una clase.

Para lograr mayor simplicidad en el ejemplo la pila se almacena en un arreglo, sin embargo, debido a la encapsulación, esta implementación puede modificarse para que la pila se almacene en una lista enlazada sin que los programas que eventualmente utilicen esta pila tengan que ser modificados.

🖫 Ejemplo 2.2: Noción de "Encapsulación" en C++

La implementación tradicional de una pila mediante un Tipo de Datos Abstracto (TDA) es la siguiente, los datos (atributos) y los procedimientos (comportamiento) se colocan por separado:

```
// Pila.hpp
// TDA Pila
#define TMP 200
typedef int TipoElemento;
typedef struct {
            int Tabla[TMP];
            int Cima;
      } TipoPila;

// Prototipos
void Inicializa_Pila(TipoPila &P);
void Meter_En_Pila(TipoPila &P, TipoElemento E);
TipoElemento Sacar_De_Pila(TipoPila &P);
int Vacia_Pila(TipoPila P);
int Llena_Pila(TipoPila P);
TipoElemento Cima_Pila(TipoPila P);
void Mostrar_Pila(TipoPila P);
```

El código de estas funciones estarían en el archivo PILA.C.

En la Programación Orientada a Objetos los datos (atributos) y procedimientos (conducta) se combinan en una única entidad llamada ***Clase***, como se muestra a continuación (para ejecutar este ejemplo cargue el proyecto E22.IDE).

```
// pila.hpp
// Definición de la Clase Pila
```

```
#define TMP 200
typedef int TipoElemento;
enum boolean {false=0, true};

class Pila{
  int Tabla[TMP];
  int Cima;
public:
  Pila();
  void InicializaPila();
  void MeterEnPila(TipoElemento E);
  TipoElemento SacarDePila();
  boolean VaciaPila();
  boolean LlenaPila();
  int CimaPila();
  void MostrarPila();
};

// Pila.CPP
// Implementación de la Clase Pila

#include <conio.h>
#include <stdio.h>
#include "pila.hpp"

Pila::Pila() {
  Cima=0;
}

void Pila::MeterEnPila(TipoElemento E) {
  Cima += 1;
  Tabla[Cima] = E;
}

TipoElemento Pila::SacarDePila() {
   Cima -= 1;
   return Tabla[Cima+1];
}

boolean Pila::VaciaPila() {
```

```
      if(Cima == 0)
    return true;
  else
    return false;
}

boolean Pila::LlenaPila() {
  if(Cima==TMP)
    return true;
  else
    return false;
}

int Pila::CimaPila() {
   return Cima;
}

void Pila::MostrarPila() {
  int i;
  clrscr();
  gotoxy(10,1);
  printf("LOS VALORES DE LA PILAS SON");
  for(i=1;i<=Cima;i++) {
       gotoxy(3*i,3);
       printf("%d",Tabla[i]);
  }
  gotoxy(5,6);
  printf("Presione una Tecla");
  getch();
}

// E22.CPP
// Programa de prueba para la Pila

#include <conio.h>
#include <stdio.h>
#include "pila.hpp"

char Menu();
void MenError(int Codigo);
void PideValor(TipoElemento &E);
```

```
void main() {
  Pila P1;
  char Op, Ch;
  TipoElemento E;
  int C;

  do {
    clrscr();
    Op=Menu();
    switch(Op) {
    case '1': if(P1.LlenaPila()==true)
                   MenError(1);
              else {
                      PideValor(E);
                     P1.MeterEnPila(E);
               }
                break;
    case '2': if(P1.VaciaPila()==true)
                      MenError(2);
              else  {
                     C=P1.SacarDePila();
                     clrscr();
                     gotoxy(4,2);
                     printf("El valor sacado es: %d",C);
                     getche();
                }
                break;
     case '3': P1.MostrarPila();
                break;
     }
  } while(Op<='3');
}

char Menu() {
  char Opcion;
  clrscr();
  gotoxy(10,2);
  printf(" ***  M E N U   P R I N C I P A L  ***");
  gotoxy(15,5);
  printf("1.  I N S E R T A R   E N   L A  P I L A");
```

```
      gotoxy(15,7);
  printf("2.  S A C A R   D E  L A   P I L A");
  gotoxy(15,9);
  printf("3.  M O S T R A R  L A  P I L A");
  gotoxy(15,11);
  printf("4.  S A L I R");
  gotoxy(20,13);
  printf("O P C I O N  [   ]");
  gotoxy(35, 13);
  Opcion=getche();
  return Opcion;
}

void MenError(int Codigo) {
  switch(Codigo) {
    case 1: clrscr();
            gotoxy(5,2);
            printf("ERROR: PILA LLENA");
         getche();
         break;
    case 2: clrscr();
            gotoxy(5,2);
            printf("ERROR: PILA VACIA ");
            getche();
            break;
  }
}

void PideValor(TipoElemento &E) {
  clrscr();
  gotoxy(2,2);
  printf(" DE EL VALOR:   ");
  scanf("%d",&E);
}
```

La noción de encapsulación existe en C++ gracias a la posibilidad de tener atributos privados, de tal modo que sentencias como las siguientes son incorrectas:

```
printf("La cima es: "%d", P1.Tabla[Cima]);
```

```
    P1.Cima++;
P1.Tabla[Cima+1]=10;
```

Estas sentencias son incorrectas dado que los atributos Tabla y Cima son ambos privados en la clase Pila.

La gran ventaja del ocultamiento de datos es que el código de los programas es completamente independiente de las estructuras de datos utilizadas. Así por ejemplo, si para implementar la Pila anterior se usa una lista enlazada en lugar del arreglo para almacenar los datos de la pila, el programa principal seguirá funcionando sin cambio alguno. Esto facilita en gran medida el mantenimiento de los programas, la reutilización de código y la modularidad de los sistemas.

2.3 Herencia de clases

La noción de herencia de clases es central en Programación Orientada a Objetos, pues es la base de la reutilización del código y de la programación incremental. La herencia permite crear una clase a partir de otra o de otras (herencia múltiple) lográndose la reutilización de clases de objetos creadas por el programador o bien creadas por otros programadores.

La herencia en C++ se basa en permitir que una clase contenga a otra clase en su declaración, la sintaxis es la siguiente:

```
class nombre_clase:  modificación_acceso  clase_base {
......
};
```

donde modificación_acceso es opcional y puede tomar como valor public o private, si el acceso es público significa que todos los miembros públicos de la clase base serán públicos en la nueva clase (clase derivada). Más adelante se explica esto con mayor detalle, por ahora utilizaremos acceso público.

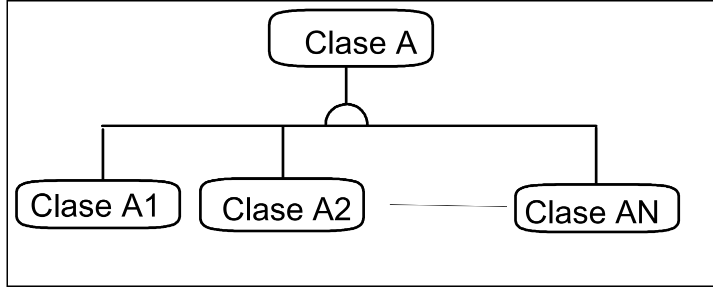
Para ilustrar la sintaxis considere las siguientes definiciones de las clases vehículo y camión.

```
class Vehiculo {
        int peso;
        int pasajeros;
public:
        void CambiaPeso(int NuevoPeso);
        int  RetornaPeso(void);
        void  CambiaPasajeros(int NuevoPeso);
        int  RetornaPasajeros(void);
}
class Camion: public  vehiculo {  // Hereda de la clase vehículo
        int carga;
public:
        void  CambiaCarga(int NuevoPeso);
        int  RetornaCarga(void);
        void  mostrar(void);
}
```

FIGURA 2.3. Ejemplo de herencia de cla-

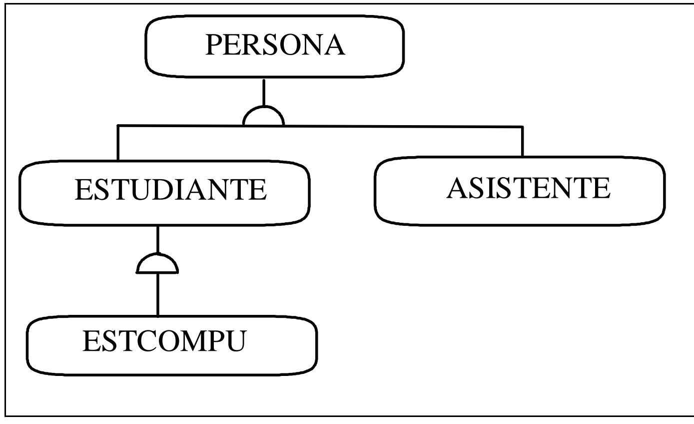
Gráficamente se representará la herencia como se muestra en la Figura 2.2, con este gráfico se representa el hecho de que las clases A1, A2,...,AN son clases *derivadas* de la clase A, llamada *clase base*. Lo cual quiere decir que clase A tiene todos los atributos y métodos comunes a las clases A1, A2,...,AN. Además el hecho de que A sea la clase base de las clases A1, A2,...,AN implica que en la clase A no debe existir ningún atributo o método que no forme parte de alguna de las clases A1, A2,...,AN. Este proceso se conoce como factorización de atributos y métodos.

FIGURA 2.2. Notación gráfica para la herencia



A continuación se presenta un ejemplo completo de herencia; las clases involucradas y su relación de herencia se presenta en el diagrama de la Figura 2.3. En esta jerarquía la clase PERSONA tiene los atributos y métodos comunes a las clases ESTUDIANTE y ASISTENTE, pero solamente aquellos que son estrictamente necesarios y comunes a ambas.

Para ejecutar este ejemplo cargue el proyecto E23.IDE.



🖫 Ejemplo 2.3: La noción de herencia en C++

```
// PERSONAS.HPP
class Persona {
protected:             // Permite que estos atributos sean

  char Nombre[30];     // conocidos en las clases derivadas
```

```
        int      Edad;
public:
   Persona(char Nom[30], int Ed);
   void Muestra(void);
   void CambiaNombre(char NuevoNom[30]);
   void CambiaEdad(int NuevaEdad);
   void ObtNombre(char Nom[30]);
   int  ObtEdad(void);
};

class Estudiante : public Persona {  // Hereda de la clase Persona
protected:
   float Examen1;
   float Examen2;
   float Promedio;
public:
   Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
   void  CalculaPromedio(void);
   void  Muestra(void);
   void  CambiaNotas(float NuevaNota1, float NuevaNota2);
   float ObtEx1(void);
   float ObtEx2(void);
   float ObtProm(void);
};

class EstCompu : public Estudiante {   // Hereda de la clase Estudiante
   float  Examen3;
public:
   EstCompu(char Nom[30],int Ed, float Ex1, float Ex2, float Ex3);
   void CalculaPromedio(void);
   float ObtEx3(void);
};

class Asistente : public Persona {   // Hereda de la clase Persona
   float Promedio;
   int HorasAs;
   char Curso[50];
   char Carrera[50];
public:
   Asistente(char Nom[30],int Ed,float Prom,char Cur[50],
                     int HA,char Carr[50]);
```

```
  void Muestra(void);
  float ObtPromedio(void);
  int  ObtHorasAs(void);
  void ObtCurso(char Cur[50]);
  void ObtCarrera(char Carr[50]);
  void CambiaPromedio(float NuevoProm);
  void CambiaHorasAs(int NuevaHo);
  void CambiaCurso(char NuevoCurso[50]);
  void CambiaCarrera(char NuevaCarrera[50]);
};
```

// Personas.cpp

```
// En este archivo va el código de las clases
#include "personas.hpp"
#include <string.h>
#include <stdio.h>

/*-----------------------P E R S O N A-----------------------------*/

Persona::Persona(char Nom[30],int Ed) {
 strcpy(Nombre,Nom);
 Edad = Ed;
}

void Persona::Muestra(void) {
 printf("Persona: %s\n",Nombre);
 printf("Edad:    %d\n",Edad);
}

void Persona::CambiaNombre(char NuevoNom[30]) {
 strcpy(Nombre,NuevoNom);
}

void Persona::CambiaEdad(int NuevaEdad) {
 Edad = NuevaEdad;
}

void Persona::ObtNombre(char Nom[30]) {
 strcpy(Nom,Nombre);
}
```

```
int Persona::ObtEdad(void) {
  return Edad;
}

/*------------------------E S T U D I A N T E---------------------------*/

// Llama al constructor de la clase base
Estudiante::Estudiante(char Nom[30],int Ed,float Ex1,float Ex2) :
                               Persona(Nom,Ed) {
  Examen1 = Ex1;
  Examen2 = Ex2;
}

void Estudiante::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2)/2;
}

void Estudiante::Muestra(void)  {
  printf("Estudiante: %s\n",Nombre);
  printf("Promedio  : %f\n",Promedio);
}

void Estudiante::CambiaNotas(float NuevaNota1, float NuevaNota2) {
  Examen1 = NuevaNota1;
  Examen2 = NuevaNota2;
}

float Estudiante::ObtEx1(void) {
  return Examen1;
}

float Estudiante::ObtEx2(void) {
  return Examen1;
}

float Estudiante::ObtProm(void) {
  return Promedio;
}

/*------------------------E S T - C O M P--------------------------*/
```

```
// Llama al constructor de la clase base
EstCompu::EstCompu(char Nom[30],int Ed,float Ex1,float Ex2,float Ex3) :                    Estudiante
   Examen3 = Ex3;
}

void EstCompu::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2 + Examen3)/3;
}

float EstCompu::ObtEx3(void) {
  return Examen3;
}

/*------------------------A S I S T E N T E-----------------------------*/

Asistente::Asistente(char Nom[30],int Ed,float Prom,char Cur[50],
                          int HA,char Carr[50]) : Persona(Nom,Ed){
   Promedio = Prom;
   strcpy(Curso,Cur);
   HorasAs = HA;
   strcpy(Carrera,Carr);
}
void Asistente::Muestra(void) {
  printf("Asistente:         %s\n",Nombre);
  printf("Edad:              %d\n",Edad);
  printf("Carrera:           %s\n",Carrera);
  printf("Horas Asignadas:   %d\n",HorasAs);
  printf("Curso Asignado:    %s\n",Curso);
  printf("Promedio Ponderado %f\n",Promedio);
}

float Asistente::ObtPromedio(void) {
  return Promedio;
}

int Asistente::ObtHorasAs(void) {
  return HorasAs;
}
```

```
    void Asistente::ObtCurso(char Cur[50]) {
  strcpy(Cur,Curso);
}

void Asistente::ObtCarrera(char Carr[50]) {
  strcpy(Carr,Carrera);
}

void Asistente::CambiaPromedio(float NuevoProm) {
  Promedio = NuevoProm;
}
void Asistente::CambiaHorasAs(int NuevaHo) {
  HorasAs = NuevaHo;
}

void Asistente::CambiaCarrera(char NuevaCarrera[50]) {
  strcpy(Carrera,NuevaCarrera);
}

void Asistente::CambiaCurso(char NuevoCurso[50]) {
  strcpy(Curso,NuevoCurso);
}
```

// E23.CPP

```
// Programa de Prueba
#include "personas.hpp"
#include <conio.h>

void main() {
   Persona P1("Carlos",20), P2("Maria",30);
   Asistente A1("Carlos",20,90,"Programación I",10,"Computación"),
              A2("María",30,80,"Cálculo 1",12,"Matemática");
   Estudiante E1("Andrea",23,70,90),
               E2("Roberto",18,70,80);
   EstCompu EC1("Karla",19,70,90,100),
             EC2("Galois",40,70,80,89);

   clrscr();
   P1.Muestra();
   P2.Muestra();
   getch();
```

```
  P1.CambiaNombre("Luis");
  P1.CambiaEdad(40);
  P1.Muestra();
  getch();

  A1.Muestra();
  A2.Muestra();
  getch();

  E1.CalculaPromedio();
  E2.CalculaPromedio();
  E1.Muestra();
  E2.Muestra();
  getch();

  EC1.CalculaPromedio();
  EC2.CalculaPromedio();
  EC1.Muestra();
  EC2.Muestra();
  getch();
}
```

FIGURA 2.4. Salida del ejemplo de herencia

La salida de este programa se muestra en la Figura 2.4.

```
C:\BC4\LIBRO_CP\E23.EXE
Persona: Carlos
Edad:    20
Persona: Maria
Edad:    30
Persona: Luis
Edad:    40
Asistente:          Carlos
Edad:               20
Carrera:            Computación
Horas Asignadas:    10
Curso Asignado:     Programación I
Promedio Ponderado 90.000000
Asistente:          Maria
Edad:               30
Carrera:            Matemática
Horas Asignadas:    12
Curso Asignado:     Cálculo 1
Promedio Ponderado 80.000000
Estudiante: Andrea
Promedio  : 80.000000
Estudiante: Roberto
Promedio  : 75.000000
```

La sintaxis para definir clases es la siguiente:

```
class nombre_clase:  modificación_acceso  clase_base {
......
};
```

donde modificación_acceso es opcional y puede tomar como valor public o private. En una clase definida con la palabra class el acceso es por defecto private, mientras que en una clase definida mediante la palabra struct el acceso es por defecto public. Ahora bien, los modificadores de acceso funcionan de acuerdo con lo siguiente:

Acceso en la clase base	**Acceso modificado**	**Acceso heredado**
public	public	público
private	public	no accesible
protected	public	protegido
public	private	privado

Acceso en la clase base	Acceso modificado	Acceso heredado
private	private	no accesible
protected	private	privado

Como ya se mencionó en C++ una clase puede heredar de dos o más clases base, esto se conoce como *herencia múltiple*. La sintaxis de la herencia múltiple es la siguiente:

```
class nombre_clase:  modificación_acceso  lista_de_clases_base {
......
};
```

Por ejemplo, en el siguiente programa la clase C hereda de la clases A y B.

```
class A {
protected:
        int a;
public:
        A(int x);
        .....
};

class B {
protected:
        int b;
public:
        B(int x);
        .....
};

class C: public A, B {      // Hereda de las clase A y B
protected:
        int c;
public:
        C(int x, int y,int z);
        .....
```

```
};
```

Un problema importante es que el constructor de una clase derivada debe invocar al constructor de la clase base. Esto se debe hacer en el mismo orden que se hizo en la declaración de la clase, de modo tal que la sintaxis del constructor es la siguiente:

```
constructor_derivado(argumentos) :    base1(argumentos),
                                      base2(argumentos), ......,
                                      baseN(argumentos) {
        ...............
}
```

Por ejemplo, para las clases A, B y C del ejemplo anterior los constructores tienen el siguiente código:

FIGURA 2.5.
Relación Componente-Compuesto de clases

```
A::A(int x) {
        a=x;
}
.......
B::B(int x) {
        b=x;
}
......
C::C(int x, int y, int z) : A(x), B(y) {
        c=z;
}
......
```

2.4 Relaciones Componente-Compuesto

Las relaciones *Componente-Compuesto* (Com-Com), llamadas así en el libro de Coad y Yourdon [Coad], indican que una clase tendrá como atributo(s) a otra(s) clase(s), es decir una clase estará compuesta de otras clases. Algunos autores se refieren a este tipo de relación como *Agregación.* En este libro usaremos el término Com-Com. La notación gráfica para éste tipo de relación se presenta en la Figura 2.5

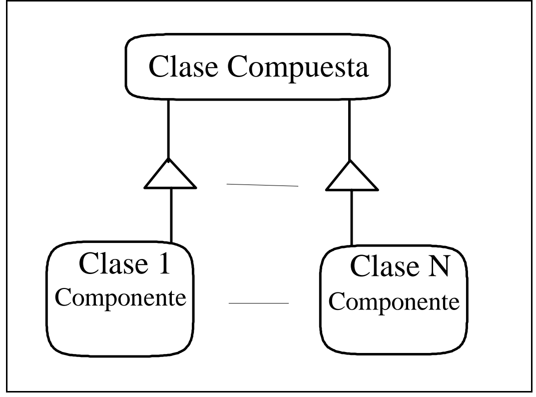
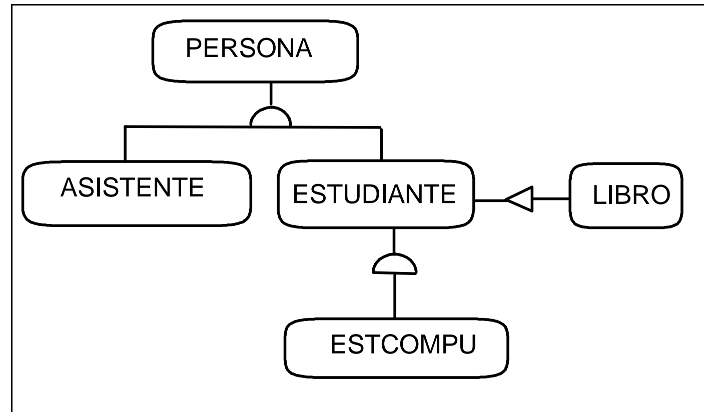


FIGURA 2.6.
Relación Com-Com

Por ejemplo, supóngase que la clase ESTUDIANTE del ejemplo 2.3 tiene como atributo una clase denominada LIBRO, modelándose así la eventual situación de que un estudiante posee un libro. Gráficamente esta relación Com-Com se representa como se muestra en la Figura 2.6.

Claramente la clase LIBRO debe tener al menos métodos que se encarguen de capturar, modificar e imprimir los datos de un libro. Ahora bien como la clase LIBRO es componente de la clase ESTUDIANTE, cuando se capturan e imprimen los datos de ESTUDIANTE se deben capturar e imprimir los datos de la clase LIBRO. La mejor forma de lograr esto es que los métodos de captura e impresión de datos de la clase ESTUDIANTE invoquen a los métodos de captura e impresión de la clase LIBRO.



Para ilustrar esta situación considere el ejemplo 2.4, en el que se hace una nueva implementación de la jerarquía de la Figura 2.6. En ésta se han agregado métodos para Captura de datos en cada una de las clases. Note el estilo de *programación incremental* utilizado en este ejemplo, es decir que cada método de Captura es implementado tomando como punto de partida la invocación del método de Captura de la clase base.

Surge una pregunta   Porqué no se implementaron en forma incremental los métodos Muestra? La respuesta es simple, no se implementaron en forma incremental pues cada método Muestra despliega un rótulo que indica qué "tipo" de persona se está desplegando, por ejemplo se despliega:

Asistente: Roberto Rojas P.

....

si se trata de una persona que es asistente. Esto hace a los métodos Muestra diferentes en cada caso.

Además se han agregado constructores sin parámetros a cada una de las clases, esta idea es importante pues en muchas situaciones una

variable o instancia de clase es declarada cuando aún no se conocen los valores de los atributos. Pero para la clase LIBRO es indispensable

implementar un constructor sin parámetros, esto porque será utilizado como componente de otra clase, en este caso de la clase ESTUDIANTE.

Para ejecutar el ejemplo 2.4 cargue el proyecto E24.IDE, analice con detalle la implementación de los métodos Muestra, Captura y de los constructores.

🖫 Ejemplo 2.4: Relación Com-Com

```
// PERS24.HPP
class Libro {
  char Nombre[30];
  int Anno;
public:
  Libro();
  void Muestra(void);
  void Captura(void);
  void CambiaNombre(char NuevoNom[30]);
  void CambiaAnno(int NuevoAnno);
  void ObtNombre(char Nom[30]);
  int  ObtAnno(void);
};

class Persona {
protected:                    // Permite que estos atributos sean
  char Nombre[30];            // conocidos en las clases derivadas
  int     Edad;
public:
  Persona(char Nom[30], int Ed);
  Persona();
  void Muestra(void);
  void  Captura(void);
  void CambiaNombre(char NuevoNom[30]);
  void CambiaEdad(int NuevaEdad);
  void ObtNombre(char Nom[30]);
  int  ObtEdad(void);
};
```

```
class Estudiante : public Persona {
protected:
   float Examen1;
   float Examen2;
   float Promedio;
   Libro L;                    // Relación Com-Com
public:
   Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
   Estudiante();
   void  CalculaPromedio(void);
   void  Muestra(void);
   void  Captura(void);
   void  CambiaNotas(float NuevaNota1, float NuevaNota2);
   float ObtEx1(void);
   float ObtEx2(void);
   float ObtProm(void);
};

class EstCompu : public Estudiante {
   float  Examen3;
   // También hereda la variable L
public:
   EstCompu(char Nom[30],int Ed, float Ex1, float Ex2, float Ex3);
   EstCompu();
   // No requiere método Muestra pues lo hereda sin ningún cambio
   void Captura(void);
   void CalculaPromedio(void);
   float ObtEx3(void);
};

class Asistente : public Persona {
   float Promedio;
   int HorasAs;
   char Curso[50];
   char Carrera[50];
public:
   Asistente(char Nom[30],int Ed,float Prom,char Cur[50],int HA,char                    Carr[50]);
   Asistente();
   void Muestra(void);
   void Captura(void);
```

```
      float ObtPromedio(void);
  int  ObtHorasAs(void);
  void ObtCurso(char Cur[50]);
  void ObtCarrera(char Carr[50]);
  void CambiaPromedio(float NuevoProm);
  void CambiaHorasAs(int NuevaHo);
  void CambiaCurso(char NuevoCurso[50]);
  void CambiaCarrera(char NuevaCarrera[50]);
};

// PERS24.CPP
#include "pers24.hpp"
#include <string.h>
#include <stdio.h>

/*------------------------P E R S O N A------------------------------*/

Persona::Persona(char Nom[30],int Ed) {
 strcpy(Nombre,Nom);
 Edad = Ed;
}

Persona::Persona() {
 strcpy(Nombre,"");
 Edad = 0;
}

void Persona::Muestra(void) {
 printf("Persona: %s\n",Nombre);
 printf("Edad:    %d\n",Edad);
}

void  Persona::Captura(void){
 printf("Nombre:    ");
 flushall();                // Evita problemas de lectura
 gets(Nombre);
 printf("Edad: ");
 scanf("%f",&Edad);
}

void Persona::CambiaNombre(char NuevoNom[30]) {
```

```
    strcpy(Nombre,NuevoNom);
}

void Persona::CambiaEdad(int NuevaEdad) {
  Edad = NuevaEdad;
}

void Persona::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}

int Persona::ObtEdad(void) {
  return Edad;
}

/*------------------------L I B R O------------------------------*/

Libro::Libro() {
  strcpy(Nombre,"");
  Anno = 0;
}

void Libro::Muestra(void) {
  printf("Nombre: %s\n",Nombre);
  printf("Año:    %d\n",Anno);
}

void Libro::Captura(void) {
  printf("Nombre del Libro: ");
  flushall();
  gets(Nombre);
  printf("Año del Libro:    ");
  scanf("%d",&Anno);
}

void Libro::CambiaNombre(char NuevoNom[30]) {
  strcpy(Nombre,NuevoNom);
}

void Libro::CambiaAnno(int NuevoAnno) {
  Anno = NuevoAnno;
```

```
   }

void Libro::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}

int Libro::ObtAnno(void) {
  return Anno;
}

/*-----------------------E S T U D I A N T E--------------------------*/

// Llama al constructor de la clase base
Estudiante::Estudiante(char Nom[30],int Ed,float Ex1,float Ex2) : Perso-
na(Nom,Ed) {
  Examen1 = Ex1;
  Examen2 = Ex2;
}

Estudiante::Estudiante() : Persona() {
  Examen1 = 0;
  Examen2 = 0;
}

void Estudiante::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2)/2;
}

void Estudiante::Muestra(void)  {
  printf("Estudiante: %s\n",Nombre);
  printf("Promedio  : %f\n",Promedio);
  L.Muestra();  // Se encarga de mostrar el Libro
}

// Programación Incremental
void Estudiante::Captura(void)  {
  Persona::Captura();           // Invoca a Captura de la clase base          printf("Nota Examen 1: ");
  scanf("%f",&Examen1);
  printf("Nota Examen 2: ");
  scanf("%f",&Examen2);
  Promedio=(Examen1+Examen2)/2;
```

```
     L.Captura();              // Invoca a Captura de la clase Libro
                               // a través de la instancia L.
}

void Estudiante::CambiaNotas(float NuevaNota1, float NuevaNota2) {
  Examen1 = NuevaNota1;
  Examen2 = NuevaNota2;
}

float Estudiante::ObtEx1(void) {
  return Examen1;
}

float Estudiante::ObtEx2(void) {
  return Examen1;
}

float Estudiante::ObtProm(void) {
  return Promedio;
}

/*------------------------E S T - C O M P---------------------------*/

// Llama al constructor de la clase base
EstCompu::EstCompu(char Nom[30],int Ed,float Ex1,float Ex2,float Ex3) : Es-
tudiante(Nom,Ed,Ex1,Ex2) {
   Examen3 = Ex3;
}

void EstCompu::Captura(void)  {
  Persona::Captura();            // Invoca a Captura de la clase Persona
  printf("Nota Examen 1: ");     // No invoca a Captura de la clase base
  scanf("%f",&Examen1);          // pues debe capturar 3 exámenes y no 2
  printf("Nota Examen 2: ");     // y porque la forma de calcular el promedio
  scanf("%f",&Examen2);          // es diferente
  printf("Nota Examen 3: ");
  scanf("%f",&Examen3);
  Promedio=(Examen1+Examen2+Examen3)/3;
  L.Captura();               // Invoca a Captura de la clase Libro
                                 // a través de la instancia L.
}
```

```
EstCompu::EstCompu() : Estudiante() {
   Examen3 = 0;
}

void EstCompu::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2 + Examen3)/3;
}

float EstCompu::ObtEx3(void) {
  return Examen3;
}

/*------------------------A S I S T E N T E-----------------------------*/

Asistente::Asistente(char Nom[30],int Ed,float Prom,char Cur[50],
                              int HA,char Carr[50]) : Persona(Nom,Ed){
   Promedio = Prom;
   strcpy(Curso,Cur);
   HorasAs = HA;
   strcpy(Carrera,Carr);
}

Asistente::Asistente(){
   Promedio = 0;
   strcpy(Curso,"");
   HorasAs = 0;
   strcpy(Carrera,"");
}

void Asistente::Muestra(void) {
  printf("Asistente:         %s\n",Nombre);
  printf("Edad:              %d\n",Edad);
  printf("Carrera:           %s\n",Carrera);
  printf("Horas Asignadas:   %d\n",HorasAs);
  printf("Curso Asignado:    %s\n",Curso);
  printf("Promedio Ponderado %f\n",Promedio);
}

void Asistente::Captura(void) {
  Persona::Captura();
```

```
    printf("Carrera:          ");
  flushall();
  gets(Carrera);
  printf("Horas Asignadas:   ");
  scanf("%d",&HorasAs);
  printf("Curso Asignado:    ");
  flushall();
  gets(Curso);
  printf("Promedio Ponderado ");
  scanf("%f",&Promedio);
}

float Asistente::ObtPromedio(void) {
  return Promedio;
}

int Asistente::ObtHorasAs(void) {
  return HorasAs;
}

void Asistente::ObtCurso(char Cur[50]) {
  strcpy(Cur,Curso);
}

void Asistente::ObtCarrera(char Carr[50]) {
  strcpy(Carr,Carrera);
}

void Asistente::CambiaPromedio(float NuevoProm) {
  Promedio = NuevoProm;
}

void Asistente::CambiaHorasAs(int NuevaHo) {
  HorasAs = NuevaHo;
}

void Asistente::CambiaCarrera(char NuevaCarrera[50]) {
  strcpy(Carrera,NuevaCarrera);
}

void Asistente::CambiaCurso(char NuevoCurso[50]) {
```

```
      strcpy(Curso,NuevoCurso);
}

// E24.CPP
// Programa de Prueba
#include "pers24.hpp"
#include <conio.h>
#include <stdio.h>

void main() {
  Libro L1;              // Variable de tipo Libro
  Estudiante E1;
  EstCompu EC1;

  L1.Captura();
  L1.Muestra();
  printf("\n\n");

  E1.Captura();          // Ejemplo de Com-Como pues
  E1.Muestra();          // pues ESTUDIANTE tiene como instancia L.
  printf("\n\n");

  EC1.Captura();         // Ejemplo de Com-Com, pues
  EC1.Muestra();         // pues ESTCOMPU hereda la instancia L.
  getch();
}
```

Es importante resaltar el método Captura de la clase ESTUDIANTE pues permite ilustrar el concepto de programación incremental:

```
void Estudiante::Captura(void)  {
  Persona::Captura();            // Relación de herencia
  printf("Nota Examen 1: ");
  scanf("%f",&Examen1);
  printf("Nota Examen 2: ");
  scanf("%f",&Examen2);
  Promedio=(Examen1+Examen2)/2;
  L.Captura();                 .        // Relación Com-Com
}
```

nótese el resaltado en negrilla de la invocación al método Captura de la clase base PERSONA (**Persona::Captura();**). También se invoca al método Captura de la clase LIBRO (**L.Captura();**), la cual es un componente de la clase ESTUDIANTE. Este estilo de programación es recomendado en Programación Orientada a Objetos, pues minimiza las líneas de código y permite la reutilización de código.

Obviamente como la clase ESTCOMPU es una clase derivada de la clase ESTUDIANTE, la clase ESTCOMPU hereda además de los atributos Nombre, Edad, Examen1, Examen2 y el atributo L de "tipo" LIBRO, por lo que también se puede capturar la información de L desde la clase ESTCOMPU, como se muestra en el siguiente fragmento de código:

```
void EstCompu::Captura(void)  {
  Persona::Captura();             // Invoca a Captura de la clase Persona
  printf("Nota Examen 1: ");      // No invoca a Captura de la clase base
  scanf("%f",&Examen1);           // pues debe capturar 3 examenes y no 2
  printf("Nota Examen 2: ");      // y porque la forma de calcular el promedio
  scanf("%f",&Examen2);           // es diferente
  printf("Nota Examen 3: ");
  scanf("%f",&Examen3);
  Promedio=(Examen1+Examen2+Examen3)/3;
  L.Captura();              // Invoca a Captura de la clase Libro
                            // a través de la instancia L.
}
```

Si un ESTUDIANTE tuviese más de un libro, entonces la relación Com-Com sería conocida como una relación Com-Com 1 a N,y se podría implementar mediante un arreglo, como se muestra a continuación en la nueva definición de la clase ESTUDIANTE:

```
class Estudiante : public Persona {
protected:
   float Examen1;
   float Examen2;
   float Promedio;
   Libro L[100];            // Declara un arreglo de 100 libros
public:
```

```
      Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
   Estudiante();
   void  CalculaPromedio(void);
   void  Muestra(void);
   void  Captura(void);
   void  CambiaNotas(float NuevaNota1, float NuevaNota2);
   float ObtEx1(void);
   float ObtEx2(void);
   float ObtProm(void);
};
```

Luego el método Captura se puede implementar utilizando sentencia for o while, como sigue:

```
void Estudiante::Captura(void)  {
  Persona::Captura();          // Invoca a Captura de la clase base          printf("Nota Examen 1: ");
  scanf("%f",&Examen1);
  printf("Nota Examen 2: ");
  scanf("%f",&Examen2);
  Promedio=(Examen1+Examen2)/2;
  for(int i=0; i<100; i++)
       L[i].Captura();
}
```

De manera similar se puede implementar el método Muestra.

Otra forma más adecuada, es implementar una clase que sea una lista enlazada de LIBROS, y declarar en la clase ESTUDIANTE una variable de este tipo. Es decir supóngase que se tiene implementada la siguiente clase:

```
class ListaLibros   {
   nodo *p_nodo;
public:
   ListaLibros();              // Constructor
   ~ListaLibros();             // Destructor
   void Captura(int ,char*);
   void Muestra();
};
```

entonces la clase ESTUDIANTE puede implementarse como sigue:

```
class Estudiante : public Persona {
protected:
   float Examen1;
   float Examen2;
   float Promedio;
   ListaLibros  LLista;   // Declara una lista enlazada de libros
public:
   Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
   Estudiante();
   void  CalculaPromedio(void);
   void  Muestra(void);
   void  Captura(void);
   void  CambiaNotas(float NuevaNota1, float NuevaNota2);
   float ObtEx1(void);
   float ObtEx2(void);
   float ObtProm(void);
};
```

Con esta implementación, los métodos Muestra y Captura de la clase ESTUDIANTE se deben implementar de modo que invoquen a los métodos Muestra y Captura de la clase LISTALIBROS, los cuales a su vez se deben implementar con base en los métodos Muestra y Captura de la clase LIBRO. Este tipo de programación incremental anidada se ilustra con mayor detalle en las siguientes secciones.

2.5 Polimorfismo, métodos virtuales y estáticos

Todos los métodos vistos hasta ahora son *estáticos*, estos se caracterizan porque las llamadas a estos métodos están determinadas en tiempo de compilación (*vinculación anticipada*).

Los métodos *virtuales* actúan diferente, pues cuando se llama a un método virtual los llamados se vinculan en tiempo de ejecución *(vinculación tardía o dinámica*).

Por ejemplo, supóngase que se desea agregar el siguiente método a la clase Persona del ejemplo 2.4:

```
void Persona::FelizCumpleanos(void) {
  Edad++;
  printf("El registro ha sido actualizado\n");
  Muestra();
}
```

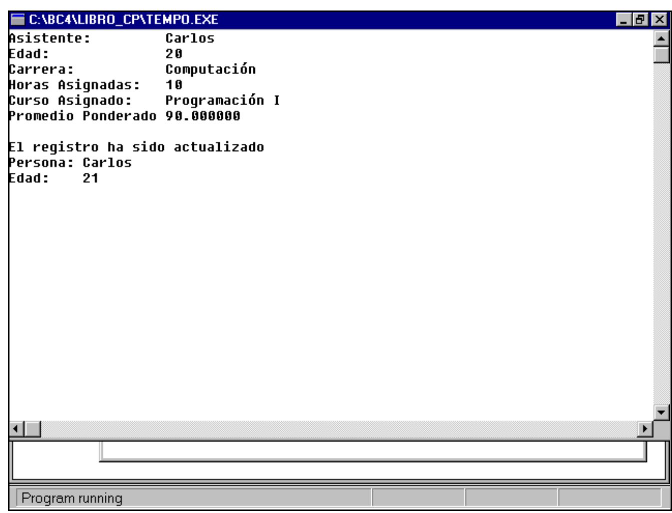
Este método actualiza la edad de una persona y luego llama al método Muestra para presentar en pantalla la actualización. La clase Asistente hereda este nuevo método puesto que es una clase derivada. A continuación se presenta un programa que utiliza este método para una instancia de la clase de "tipo" Asistente:

FIGURA 2.7. Salida del método: Feliz-

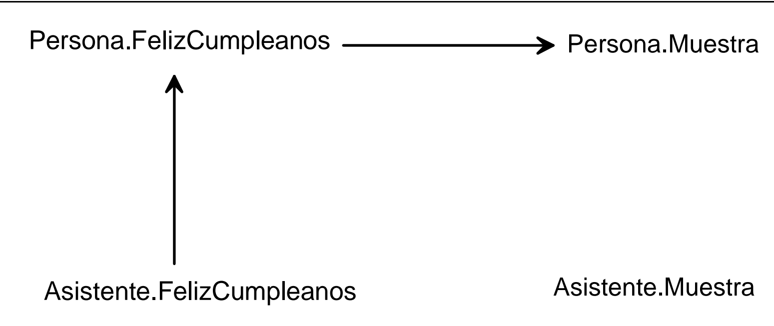
```
#include <conio.h>
#include <stdio.h>
#include "pers24.hpp"

void main() {
  Asistente A1("Carlos",20,90,"Programación I",10,"Computación");
  A1.Muestra();                    // Llamado a Muestra()
  printf("\n");
  A1.FelizCumpleanos();            // Llamado a Muestra()
  getch();
}
```

Este programa produce la siguiente salida de la Figura 2.7.



Nótese que esta salida es incorrecta, pues cuando se llama al método Muestra a través del método FelizCumpleanos se invoca al método Muestra de la clase Persona, cuando se debería llamar al método Muestra de la clase Asistente. En el siguiente gráfico se ilustra lo sucedido:



Una posible solución a esta deficiencia es redefinir el método FelizCumpleanos en la clase Asistente, pero esto va contra la filosofía de la Programación Orientada a Objetos (POO) que trata de reducir el código redundante. La solución de la POO es el uso de *métodos virtuales*.

C++ requiere de dos pasos para convertir un método en virtual.

1. Crear un constructor de la clase (si este no ha sido creado).
2. Agregar la palabra reservada virtual en la parte de definición de la clase.

A continuación se presentan las definiciones de las clases Persona y Asistente corregidas:

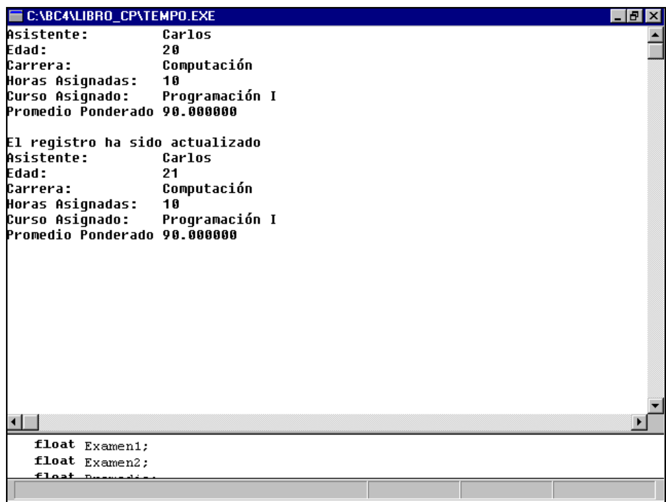
```
class Persona {
protected:                  // Permite que estos atributos sean
  char Nombre[30];          // conocidos en las clases derivadas
  int    Edad;
public:
  Persona(char Nom[30], int Ed);
  virtual void Muestra(void);
  void CambiaNombre(char NuevoNom[30]);
  void CambiaEdad(int NuevaEdad);
  void ObtNombre(char Nom[30]);
  int  ObtEdad(void);
  void FelizCumpleanos(void);
};
..............................
```

FIGURA 2.8. Salida del método Fe-

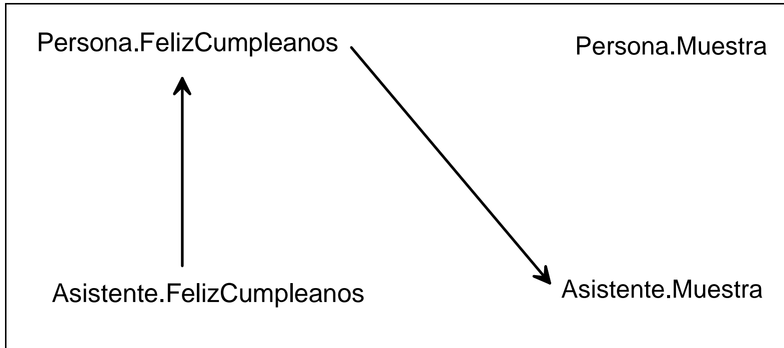
```
class Asistente : public Persona {
  float Promedio;
  int HorasAs;
  char Curso[50];
  char Carrera[50];
public:
  Asistente(char Nom[30],int Ed,float Prom,char Cur[50],
                int HA,char Carr[50]);
  virtual void Muestra(void);
  float ObtPromedio(void);
  int  ObtHorasAs(void);
  void ObtCurso(char Cur[50]);
  void ObtCarrera(char Carr[50]);
  void CambiaPromedio(float NuevoProm);
  void CambiaHorasAs(int NuevaHo);
```

```
      void CambiaCurso(char NuevoCurso[50]);
  void CambiaCarrera(char NuevaCarrera[50]);
};
```

Una vez hecha esta corrección la salida del programa es la que se muestra en la Figura 2.8, la cual ahora es correcta.



La declaración del constructor es necesaria al usar métodos virtuales, pues este inicializa la *Tabla de Métodos Virtuales TVM*. El siguiente gráfico ilustra lo sucedido:



Compatibilidad entre objetos

Las variables "tipo" clase siguen reglas de compatibilidad un poco diferentes a las variables normales de C. La diferencia básica es que una variable de un tipo es compatible con cualquier otra variable de un tipo descendiente en la jerarquía, pero no a la inversa. Por ejemplo, el siguiente programa es correcto (con la definición de Persona y Asistente del ejemplo 2.4):

```
#include "pers24.hpp"
#include <conio.h>
#include <stdio.h>

void main() {
  Persona P1("Carlos",20);
  Asistente A1("Adrea",19,90,"Programación I",10,"Computación");
  P1=A1;
  P1.Muestra();
 }
```

es válida pues a P1 se le asignan todos los campos comunes con A1 y los demás campos se desestiman. Pero el siguiente programa produce error de compilación:

```
#include "pers24.hpp"
#include <conio.h>
#include <stdio.h>
```

```
void main() {
  Persona P1("Carlos",20);
  Asistente A1("Adrea",19,90,"Programación I",10,"Computación");
  A1=P1;
  A1.Muestra();
}
```

pues el compilador no sabe qué hacer con los campos o atributos adicionales de la clase Asistente respecto de la clase Persona.

La flexibilidad de la compatibilidad de tipos permite un concepto potente en POO, *el polimorfismo*, es decir que un procedimiento o función está dispuesto a aceptar un amplio rango de tipos como parámetro, los cuales incluso se pueden determinar en tiempo de ejecución.

Además esto permite agregar otro concepto importante en POO, *la extensibilidad*, la cual permite a los programadores trabajar con módulos ya compilados y agregar sus propias clases derivadas, tal como sucede con *Object-Windows*.

Punteros a Objetos

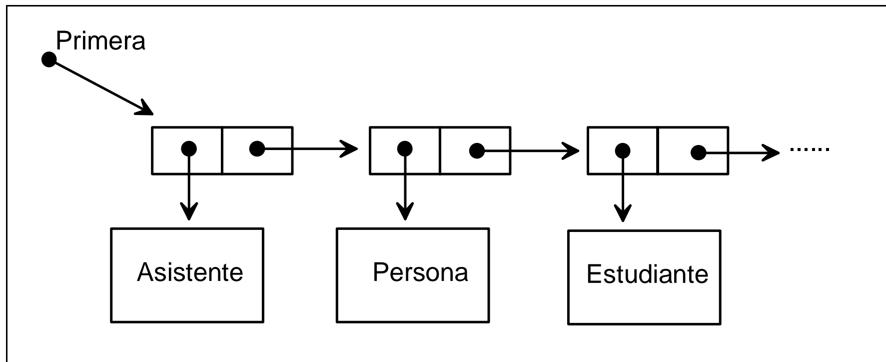
C++ soporta la creación de punteros a clases, por ejemplo el siguiente programa usa una variable tipo puntero a una Persona:

```
#include "pers24.hpp"
#include <conio.h>
#include <stdio.h>

void main() {
  Persona *PP1;
  PP1=new Persona("Rolando",10);// Pide memoria y llama al constructor
  PP1->Muestra();                   // Invoca el método Muestra
  PP1->CambiaEdad(22);              // Invoca el método CambiaEdad
  PP1->Muestra();
  delete(PP1);                      // Libera la memoria reservada con new
}
```

Del fragmento de código anterior es importante resaltar el hecho de que se puede usar el operador -> (flecha) para invocar métodos de una clase, extendiendo las propiedades que ya tenía este operador en las estructuras del lenguaje C.

En el siguiente ejemplo con la ayuda de los punteros a clases y de los métodos virtuales se implementará una *lista enlazada polimórfica*[3] de nodos "tipo" Persona, Asistente, Estudiante y EstCompu definidos en el ejemplo 2.4. Gráficamente esta lista se puede representar como sigue:



🖫 Ejemplo 2.5: La noción de polimorfismo en C++ en una lista

```
// PERS25.HPP
// protege para múltiple inclusión
#if ! defined(_I_PERS25_HPP_)
#define _I_PERS25_HPP_

class Libro {
   char Nombre[30];
   int Anno;
public:
   Libro();
   virtual void Muestra(void);
   virtual void Captura(void);
   virtual void CambiaNombre(char NuevoNom[30]);
```

1 [3] El término polimorfismo combina las palabras en Latín "muchas" y "formas".

```
      virtual void CambiaAnno(int NuevoAnno);
   virtual void ObtNombre(char Nom[30]);
   virtual int  ObtAnno(void);
};

class Persona {
protected:                // Permite que estos atributos sean
   char Nombre[30];       // conocidos en las clases derivadas
   int     Edad;
public:
   Persona(char Nom[30], int Ed);
   Persona();
   virtual void Muestra(void);
   virtual void Captura(void);
   virtual void CambiaNombre(char NuevoNom[30]);
   virtual void CambiaEdad(int NuevaEdad);
   virtual void ObtNombre(char Nom[30]);
   virtual int  ObtEdad(void);
};

class Estudiante : public Persona {
protected:
   float Examen1;
   float Examen2;
   float Promedio;
   Libro L;               // Relación Com-Com
public:
   Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
   Estudiante();
   virtual void  CalculaPromedio(void);
   virtual void  Muestra(void);
   virtual void  Captura(void);
   void  CambiaNotas(float NuevaNota1, float NuevaNota2);
   virtual float ObtEx1(void);
   virtual float ObtEx2(void);
   virtual float ObtProm(void);
};

class EstCompu : public Estudiante {
   float  Examen3;
public:
```

```
      EstCompu(char Nom[30],int Ed, float Ex1, float Ex2, float Ex3);
  EstCompu();
  // No requiere método Muestra pues lo hereda sin ningún cambio
  virtual void Captura(void);
  void CambiaNotas(float NuevaNota1, float NuevaNota2, float                    NuevaNota
  virtual void CalculaPromedio(void);
  virtual float ObtEx3(void);
};

class Asistente : public Persona {
  float Promedio;
  int HorasAs;
  char Curso[50];
  char Carrera[50];
public:
  Asistente(char Nom[30],int Ed,float Prom,char Cur[50],int HA,char            Carr[50]);
  Asistente();
  virtual void Muestra(void);
  virtual void Captura(void);
  virtual float ObtPromedio(void);
  virtual int  ObtHorasAs(void);
  virtual void ObtCurso(char Cur[50]);
  virtual void ObtCarrera(char Carr[50]);
  virtual void CambiaPromedio(float NuevoProm);
  virtual void CambiaHorasAs(int NuevaHo);
  virtual void CambiaCurso(char NuevoCurso[50]);
  virtual void CambiaCarrera(char NuevaCarrera[50]);
};
#endif

// PERS25.CPP
#include "pers25.hpp"
#include <string.h>
#include <stdio.h>

/*-----------------------P E R S O N A-----------------------------*/

Persona::Persona(char Nom[30],int Ed) {
 strcpy(Nombre,Nom);
 Edad = Ed;
}
```

```
Persona::Persona() {
  strcpy(Nombre,"");
  Edad = 0;
}

void Persona::Muestra(void) {
  printf("Persona: %s\n",Nombre);
  printf("Edad:    %d\n",Edad);
}

void  Persona::Captura(void){
  printf("Nombre:    ");
  flushall();
  gets(Nombre);
  printf("Edad: ");
  scanf("%d",&Edad);
}

void Persona::CambiaNombre(char NuevoNom[30]) {
  strcpy(Nombre,NuevoNom);
}

void Persona::CambiaEdad(int NuevaEdad) {
  Edad = NuevaEdad;
}

void Persona::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}

int Persona::ObtEdad(void) {
  return Edad;
}

/*------------------------L I B R O------------------------------*/

Libro::Libro() {
  strcpy(Nombre,"");
  Anno = 0;
}
```

```
void Libro::Muestra(void) {
  printf("Nombre del libro: %s\n",Nombre);
  printf("Año del libro:    %d\n",Anno);
}

void Libro::Captura(void) {
  printf("Nombre del Libro: ");
  flushall();
  gets(Nombre);
  printf("Año del Libro:    ");
  scanf("%d",&Anno);
}

void Libro::CambiaNombre(char NuevoNom[30]) {
  strcpy(Nombre,NuevoNom);
}

void Libro::CambiaAnno(int NuevoAnno) {
  Anno = NuevoAnno;
}

void Libro::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}

int Libro::ObtAnno(void) {
  return Anno;
}

/*-----------------------E S T U D I A N T E--------------------------*/

// Llama al constructor de la clase base
Estudiante::Estudiante(char Nom[30],int Ed,float Ex1,float Ex2) : Perso-
na(Nom,Ed) {
  Examen1 = Ex1;
  Examen2 = Ex2;
}

Estudiante::Estudiante() : Persona() {
  Examen1 = 0;
```

```
        Examen2 = 0;
}

void Estudiante::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2)/2;
}

void Estudiante::Muestra(void)  {
  printf("Estudiante: %s\n",Nombre);
  printf("Promedio  : %f\n",Promedio);
  L.Muestra();  // Se encarga de mostrar el Libro
}

// Programación Incremental
void Estudiante::Captura(void)  {
  Persona::Captura();                  // Invoca a Captura de la clase base
  printf("Nota Examen 1: ");
  scanf("%f",&Examen1);
  printf("Nota Examen 2: ");
  scanf("%f",&Examen2);
  Promedio=(Examen1+Examen2)/2;
  L.Captura();                  // Invoca a Captura de la clase Libro
                                // a través de la instancia L.
}

void Estudiante::CambiaNotas(float NuevaNota1, float NuevaNota2) {
  Examen1 = NuevaNota1;
  Examen2 = NuevaNota2;
}

float Estudiante::ObtEx1(void) {
  return Examen1;
}

float Estudiante::ObtEx2(void) {
  return Examen1;
}

float Estudiante::ObtProm(void) {
  return Promedio;
}
```

```
/*-----------------------E S T - C O M P--------------------------*/

// Llama al constructor de la clase base
EstCompu::EstCompu(char Nom[30],int Ed,float Ex1,float Ex2,float Ex3) : Es-
tudiante(Nom,Ed,Ex1,Ex2) {
   Examen3 = Ex3;
}

void EstCompu::Captura(void)  {
  Persona::Captura();              // Invoca a Captura de la clase Persona
  printf("Nota Examen 1: ");       // No invoca a Captura de la clase base
  scanf("%f",&Examen1);            // pues debe capturar 3 exámenes y no 2
  printf("Nota Examen 2: ");       // y porque la forma de calcular el promedio
  scanf("%f",&Examen2);            // es diferente
  printf("Nota Examen 3: ");
  scanf("%f",&Examen3);
  Promedio=(Examen1+Examen2+Examen3)/3;
  L.Captura();                // Invoca a Captura de la clase Libro
                              // a través de la instancia L.
}

EstCompu::EstCompu() : Estudiante() {
   Examen3 = 0;
}

void EstCompu::CambiaNotas(float NuevaNota1, float NuevaNota2, float Nue-
vaNota3) {
  Examen1 = NuevaNota1;
  Examen2 = NuevaNota2;
  Examen3 = NuevaNota3;
}


void EstCompu::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2 + Examen3)/3;
}

float EstCompu::ObtEx3(void) {
  return Examen3;
}
```

```
/*-----------------------A S I S T E N T E----------------------------*/

Asistente::Asistente(char Nom[30],int Ed,float Prom,char Cur[50],
                         int HA,char Carr[50]) : Persona(Nom,Ed){
   Promedio = Prom;
   strcpy(Curso,Cur);
   HorasAs = HA;
   strcpy(Carrera,Carr);
}

Asistente::Asistente(){
   Promedio = 0;
   strcpy(Curso,"");
   HorasAs = 0;
   strcpy(Carrera,"");
}

void Asistente::Muestra(void) {
  printf("Asistente:         %s\n",Nombre);
  printf("Edad:              %d\n",Edad);
  printf("Carrera:           %s\n",Carrera);
  printf("Horas Asignadas:   %d\n",HorasAs);
  printf("Curso Asignado:    %s\n",Curso);
  printf("Promedio Ponderado %f\n",Promedio);
}

void Asistente::Captura(void) {
  Persona::Captura();
  printf("Carrera:           ");
  flushall();
  gets(Carrera);
  printf("Horas Asignadas:   ");
  scanf("%d",&HorasAs);
  printf("Curso Asignado:    ");
  flushall();
  gets(Curso);
  printf("Promedio Ponderado ");
  scanf("%f",&Promedio);
}
```

```
    float Asistente::ObtPromedio(void) {
  return Promedio;
}

int Asistente::ObtHorasAs(void) {
  return HorasAs;
}

void Asistente::ObtCurso(char Cur[50]) {
  strcpy(Cur,Curso);
}

void Asistente::ObtCarrera(char Carr[50]) {
  strcpy(Carr,Carrera);
}

void Asistente::CambiaPromedio(float NuevoProm) {
  Promedio = NuevoProm;
}

void Asistente::CambiaHorasAs(int NuevaHo) {
  HorasAs = NuevaHo;
}

void Asistente::CambiaCarrera(char NuevaCarrera[50]) {
  strcpy(Carrera,NuevaCarrera);
}

void Asistente::CambiaCurso(char NuevoCurso[50]) {
  strcpy(Curso,NuevoCurso);
}

// LPERS25.HPP
// protege para múltiple inclusión
#if ! defined(_I_LPERS25_HPP_)
#define _I_LPERS25_HPP_

#include "pers25.hpp"

struct Nodo {
   Persona *PPersona;
```

```
        Nodo    *Sig;
};

class ListaPersonas {
  Nodo *Primera;
public:
  ListaPersonas();
  ~ListaPersonas();
  void Inserta(Persona *NuevaPersona);
  void Muestra(void);
};
#endif


// LPERS25.CPP
#include <stdio.h>
#include "lpers25.hpp"
#include "pers25.hpp"

ListaPersonas::ListaPersonas() {
   Primera = NULL;
}

ListaPersonas::~ListaPersonas() {
   Nodo *Tempo=Primera;
   while(Primera != NULL) {
      Primera=Primera->Sig;
      delete Tempo->PPersona;
      delete Tempo;
   }
}

void ListaPersonas::Inserta(Persona *NuevaPersona) {
   Nodo *NodoTemp, *NuevoNodo;
   NuevoNodo = new Nodo;
   NuevoNodo->PPersona = NuevaPersona;
   NuevoNodo->Sig = NULL;
   if(Primera==NULL)
      Primera = NuevoNodo;
   else {
        NodoTemp=Primera;
```

```
            while(NodoTemp->Sig != NULL)
          NodoTemp=NodoTemp->Sig;
        NodoTemp->Sig=NuevoNodo;
  }
}

void ListaPersonas::Muestra() {
  Nodo *Actual=Primera;
  while(Actual != NULL) {
       Actual->PPersona->Muestra();
       printf("\n");
       Actual = Actual->Sig;
  }
}

// Programa de Prueba
// E25.CPP
#include "pers25.hpp"
#include "lpers25.hpp"
#include <conio.h>
#include <stdio.h>

void LeeDatosPersona(void);
void LeeDatosAsistente(void);
void LeeDatosEstudiante(void);
void LeeDatosEstCompu(void);
char Menu(void);
Asistente *PAsistente;
Persona *PPersona;
Estudiante *PEstudiante;
EstCompu *PEstCompu;

void main() {
  ListaPersonas UnaLista;
  char opcion;
  do {
    clrscr();
    opcion = Menu();
    switch(opcion) {
      case '1': clrscr();
                    LeeDatosPersona();
```

```
                        UnaLista.Inserta(PPersona);
                    break;
      case '2': clrscr();
                     LeeDatosAsistente();
                    UnaLista.Inserta(PAsistente);
                    break;
      case '3': clrscr();
                    LeeDatosEstudiante();
                    UnaLista.Inserta(PEstudiante);
                    break;
      case '4': clrscr();
                    LeeDatosEstCompu();
                    UnaLista.Inserta(PEstCompu);
                    break;
      case '5': clrscr();
                    UnaLista.Muestra();
                    getch();
                    break;
    }
  } while(opcion!='6');
}

void LeeDatosPersona() {
  PPersona=new Persona();
  PPersona->Captura();
}

void LeeDatosEstudiante() {
  PEstudiante=new Estudiante();
  PEstudiante->Captura();
}

void LeeDatosEstCompu() {
  PEstCompu=new EstCompu();
  PEstCompu->Captura();
}

void LeeDatosAsistente() {
  PAsistente = new Asistente();
  PAsistente->Captura();
}
```

```
char Menu(void) {
    char opcion;
    printf("[1] Insertar Persona\n");
    printf("[2] Insertar Asistente\n");
    printf("[3] Insertar Estudiante\n");
    printf("[4] Insertar Estudiante de Computación\n");
    printf("[5] Ver la Lista Polimórfica\n");
    printf("[6] Salir\n");
    opcion= getch();
    return opcion;
}
```

Es importante destacar las definiciones de las clases Estudiante y EstCompu que se presentan a continuación:

```
class Estudiante : public Persona {
protected:
   float Examen1;
   float Examen2;
   float Promedio;
   Libro L;                    // Relación Com-Com
public:
   Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
   Estudiante();
   virtual void  Muestra(void);
   virtual void  Captura(void);
   void  CambiaNotas(float NuevaNota1, float NuevaNota2);
   virtual void  CalculaPromedio(void);
   virtual float ObtEx1(void);
   virtual float ObtEx2(void);
   virtual float ObtProm(void);
};

class EstCompu : public Estudiante {
   float  Examen3;
public:
   EstCompu(char Nom[30],int Ed, float Ex1, float Ex2, float Ex3);
   EstCompu();
   // No requiere método Muestra pues lo hereda sin ningún cambio
```

```
      virtual void Captura(void);
   void CambiaNotas(float NuevaNota1, float NuevaNota2, float                NuevaNot
   virtual void CalculaPromedio(void);
   virtual float ObtEx3(void);
};
```

En el código anterior de definen algunos métodos como virtuales mediante el uso de la palabra reservada ***virtual***. Es importante aclarar que cuando un método se declara virtual en la clase base entonces será virtual en todas sus clases derivadas sin necesidad de declararlos de nuevo virtuales (aunque no es error hacerlo).

FIGURA 2.9. Lista Polimórfica de Personas

Surge una pregunta interesante de la definición de las clases Estudiante y EstCompu: Por qué el método CambiaNotas no se definió como virtual? La respuesta es simple, para que un método sea virtual en una jerarquía de clases debe tener el mismo prototipo en todas las clases en que es definido, es decir debe tener el mismo número y tipo de parámetros. Esto tiene pleno sentido ya que cuando dos métodos tienen el mismo nombre pero diferente número o tipo de parámetros, C++ considera que es una sobrecarga de funciones, por lo que no necesita que sean declarados virtuales para decidir cual método invoca, dado que lo puede determinar por el número de parámetros o bien por el tipo de éstos.

Una posible salida de este programa es la lista polimórfica que se muestra en la Figura 2.9.

```
C:\BC4\LIBRO_CP\E25.EXE
Persona: Bernal Lizano
Edad:    33

Asistente:          Andrea Rojas
Edad:               25
Carrera:            Computación
Horas Asignadas:    12
Curso Asignado:     Programación I
Promedio Ponderado 90.000000

Estudiante: Damaris Céspedes
Promedio   : 90.000000
Nombre del libro: Cálculo I
Año del libro:     1990

Estudiante: Pedro Méndez
Promedio   : 85.666664
Nombre del libro: Programación C++
Año del libro:     1996
```

La salida de la Figura 2.9 es producida por la invocación del método Muestra de la clase ListaPersonas, el cual tiene el siguiente código:

```
void ListaPersonas::Muestra() {
  Nodo *Actual=Primera;
  while(Actual != NULL) {
     Actual->PPersona->Muestra();     // Llamado a métodos virtuales
     printf("\n");
     Actual = Actual->Sig;
  }
}
```

mediante la instrucción ***Actual->PPersona->Muestra();*** se logra que en tiempo de ejecución C++ invoque al método Muestra adecuado, como puede se observar en la Figura 2.9, gracias a que los métodos Muestra han sido definidos virtuales.

A continuación se presenta nuevamente el ejemplo 2.5, pero ahora se agregan nuevos métodos virtuales que permiten Guardar y Recuperar en un archivo binario objetos "tipo" Persona, Estudiante, EstCompu y Asistente, para luego agregar los métodos que permiten Guardar y Recu-

perar la lista polimórfica completa en un archivo. Los aspectos nuevos en este ejemplo se resaltan en letras tipo negrilla e itálica. Para ejecutar el ejemplo 2.6 cargue el proyecto E26.IDE.

🖫 Ejemplo 2.6: La noción de polimorfismo en un archivo

```
// PERS26.HPP
// protege para múltiple inclusión
#if ! defined(_I_PERS26_HPP_)
#define _I_PERS26_HPP_
#include <stdio.h>
#define MAX1 30
#define MAX2 50

class Libro {
   char Nombre[30];
   int Anno;
public:
   Libro();
   virtual void Muestra(void);
   virtual void Captura(void);
   virtual void CambiaNombre(char NuevoNom[30]);
   virtual void CambiaAnno(int NuevoAnno);
   virtual void ObtNombre(char Nom[30]);
   virtual int  ObtAnno(void);
   virtual void Guardar(FILE *archivo);
   virtual void Recuperar(FILE *archivo);
};

class Persona {
protected:                // Permite que estos atributos sean
   char Nombre[30];       // conocidos en las clases derivadas
   int     Edad;
public:
   Persona(char Nom[30], int Ed);
   Persona();
   virtual void Muestra(void);
   virtual void Captura(void);
   virtual void CambiaNombre(char NuevoNom[30]);
   virtual void CambiaEdad(int NuevaEdad);
   virtual void ObtNombre(char Nom[30]);
```

```
      virtual int  ObtEdad(void);
   virtual void Guardar(FILE *archivo);
   virtual void Recuperar(FILE *archivo);
};

class Estudiante : public Persona {
protected:
   float Examen1;
   float Examen2;
   float Promedio;
   Libro L;                    // Relación Com-Com
public:
   Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
   Estudiante();
   virtual void  CalculaPromedio(void);
   virtual void  Muestra(void);
   virtual void  Captura(void);
   void  CambiaNotas(float NuevaNota1, float NuevaNota2);
   virtual float ObtEx1(void);
   virtual float ObtEx2(void);
   virtual float ObtProm(void);
   virtual void Guardar(FILE *archivo);
   virtual void Recuperar(FILE *archivo);
};

class EstCompu : public Estudiante {
   float  Examen3;
public:
   EstCompu(char Nom[30],int Ed, float Ex1, float Ex2, float Ex3);
   EstCompu();
   // No requiere método Muestra pues lo hereda sin ningún cambio
   virtual void Captura(void);
   void CambiaNotas(float NuevaNota1, float NuevaNota2, float NuevaNota3);
   virtual void CalculaPromedio(void);
   virtual float ObtEx3(void);
   virtual void Guardar(FILE *archivo);
   virtual void Recuperar(FILE *archivo);
};

class Asistente : public Persona {
   float Promedio;
```

```
        int HorasAs;
   char Curso[50];
   char Carrera[50];
public:
   Asistente(char Nom[30],int Ed,float Prom,char Cur[50],int HA,char Carr[50]);
   Asistente();
   virtual void Muestra(void);
   virtual void Captura(void);
   virtual float ObtPromedio(void);
   virtual int  ObtHorasAs(void);
   virtual void ObtCurso(char Cur[50]);
   virtual void ObtCarrera(char Carr[50]);
   virtual void CambiaPromedio(float NuevoProm);
   virtual void CambiaHorasAs(int NuevaHo);
   virtual void CambiaCurso(char NuevoCurso[50]);
   virtual void CambiaCarrera(char NuevaCarrera[50]);
   virtual void Guardar(FILE *archivo);
   virtual void Recuperar(FILE *archivo);
};
#endif

// PERS26.CPP
#include "pers26.hpp"
#include <string.h>
#include <stdio.h>

/*------------------------P E R S O N A------------------------------*/

Persona::Persona(char Nom[30],int Ed) {
  strcpy(Nombre,Nom);
  Edad = Ed;
}

Persona::Persona() {
  strcpy(Nombre,"");
  Edad = 0;
}

void Persona::Muestra(void) {
  printf("Persona: %s\n",Nombre);
  printf("Edad:    %d\n",Edad);
```

```
    }

void  Persona::Captura(void){
  printf("Nombre:    ");
  flushall();
  gets(Nombre);
  printf("Edad: ");
  scanf("%d",&Edad);
}

void Persona::CambiaNombre(char NuevoNom[30]) {
  strcpy(Nombre,NuevoNom);
}

void Persona::CambiaEdad(int NuevaEdad) {
  Edad = NuevaEdad;
}

void Persona::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}

int Persona::ObtEdad(void) {
  return Edad;
}

void Persona::Guardar(FILE *archivo) {
  char aux='P';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
}

void Persona::Recuperar(FILE *archivo) {
  fread(&Edad,sizeof(int),1,archivo);
  fread(Nombre,MAX1*sizeof(char),1,archivo);
}

/*-----------------------L I B R O-----------------------------*/

Libro::Libro() {
```

```
    strcpy(Nombre,"");
  Anno = 0;
}

void Libro::Muestra(void) {
  printf("Nombre del libro: %s\n",Nombre);
  printf("Año del libro:    %d\n",Anno);
}

void Libro::Captura(void) {
  printf("Nombre del Libro: ");
  flushall();
  gets(Nombre);
  printf("Año del Libro:    ");
  scanf("%d",&Anno);
}

void Libro::CambiaNombre(char NuevoNom[30]) {
  strcpy(Nombre,NuevoNom);
}

void Libro::CambiaAnno(int NuevoAnno) {
  Anno = NuevoAnno;
}

void Libro::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}

int Libro::ObtAnno(void) {
  return Anno;
}

void Libro::Guardar(FILE *archivo) {
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Anno,sizeof(int),1,archivo);
}

void Libro::Recuperar(FILE *archivo) {
  fread(Nombre,MAX1*sizeof(char),1,archivo);
  fread(&Anno,sizeof(int),1,archivo);
```

```
}

/*-----------------------E S T U D I A N T E--------------------------*/

// Llama al constructor de la clase base
Estudiante::Estudiante(char  Nom[30],int  Ed,float  Ex1,float  Ex2)  :  Perso-
na(Nom,Ed) {
 Examen1 = Ex1;
 Examen2 = Ex2;
}

Estudiante::Estudiante() : Persona() {
 Examen1 = 0;
 Examen2 = 0;
}

void Estudiante::CalculaPromedio(void) {
 Promedio = (Examen1 + Examen2)/2;
}

void Estudiante::Muestra(void)  {
 printf("Estudiante: %s\n",Nombre);
 printf("Promedio  : %f\n",Promedio);
 L.Muestra();  // Se encarga de mostrar el Libro
}

// Programación Incremental
void Estudiante::Captura(void)  {
 Persona::Captura();               // Invoca a Captura de la clase base                    printf("Nota Examen 1: ");
 scanf("%f",&Examen1);
 printf("Nota Examen 2: ");
 scanf("%f",&Examen2);
 Promedio=(Examen1+Examen2)/2;
 L.Captura();               // Invoca a Captura de la clase Libro
                            // a través de la instancia L.
}

void Estudiante::CambiaNotas(float NuevaNota1, float NuevaNota2) {
 Examen1 = NuevaNota1;
 Examen2 = NuevaNota2;
}
```

```
float Estudiante::ObtEx1(void) {
  return Examen1;
}

float Estudiante::ObtEx2(void) {
  return Examen1;
}

float Estudiante::ObtProm(void) {
  return Promedio;
}

void Estudiante::Guardar(FILE *archivo) {
  char aux='E';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Examen1,sizeof(float),1,archivo);
  fwrite(&Examen2,sizeof(float),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
  L.Guardar(archivo);                          // Relación Com-Com
}

void Estudiante::Recuperar(FILE *archivo) {
  Persona::Recuperar(archivo);
  fread(&Examen1,sizeof(float),1,archivo);
  fread(&Examen2,sizeof(float),1,archivo);
  fread(&Promedio,sizeof(float),1,archivo);
  L.Recuperar(archivo);                        // Relación Com-Com
}

/*------------------------E S T - C O M P---------------------------*/

// Llama al constructor de la clase base
EstCompu::EstCompu(char Nom[30],int Ed,float Ex1,float Ex2,float Ex3) : Es-
tudiante(Nom,Ed,Ex1,Ex2) {
   Examen3 = Ex3;
}

void EstCompu::Captura(void)  {
```

```
     Persona::Captura();              // Invoca a Captura de la clase Persona
  printf("Nota Examen 1: ");       // No invoca a Captura de la clase base
  scanf("%f",&Examen1);            // pues debe capturar 3 examenes y no 2
  printf("Nota Examen 2: ");       // y porque la forma de calcular el promedio
  scanf("%f",&Examen2);            // es diferente
  printf("Nota Examen 3: ");
  scanf("%f",&Examen3);
  Promedio=(Examen1+Examen2+Examen3)/3;
  L.Captura();               // Invoca a Captura de la clase Libro
                             // a través de la instancia L.
}

EstCompu::EstCompu() : Estudiante() {
   Examen3 = 0;
}

void EstCompu::CambiaNotas(float NuevaNota1, float NuevaNota2, float Nue-
vaNota3) {
  Examen1 = NuevaNota1;
  Examen2 = NuevaNota2;
  Examen3 = NuevaNota3;
}

void EstCompu::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2 + Examen3)/3;
}

float EstCompu::ObtEx3(void) {
  return Examen3;
}

void EstCompu::Guardar(FILE *archivo) {
  char aux='C';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Examen1,sizeof(float),1,archivo);
  fwrite(&Examen2,sizeof(float),1,archivo);
  fwrite(&Examen3,sizeof(float),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
  L.Guardar(archivo);
```

```
   }

void EstCompu::Recuperar(FILE *archivo) {
  Persona::Recuperar(archivo);
  fread(&Examen1,sizeof(float),1,archivo);
  fread(&Examen2,sizeof(float),1,archivo);
  fread(&Examen3,sizeof(float),1,archivo);
  fread(&Promedio,sizeof(float),1,archivo);
  L.Recuperar(archivo);
}

/*------------------------A S I S T E N T E-----------------------------*/

Asistente::Asistente(char Nom[30],int Ed,float Prom,char Cur[50],
                     int HA,char Carr[50]) : Persona(Nom,Ed){
   Promedio = Prom;
   strcpy(Curso,Cur);
   HorasAs = HA;
   strcpy(Carrera,Carr);
}

Asistente::Asistente(){
   Promedio = 0;
   strcpy(Curso,"");
   HorasAs = 0;
   strcpy(Carrera,"");
}

void Asistente::Muestra(void) {
  printf("Asistente:         %s\n",Nombre);
  printf("Edad:              %d\n",Edad);
  printf("Carrera:           %s\n",Carrera);
  printf("Horas Asignadas:   %d\n",HorasAs);
  printf("Curso Asignado:    %s\n",Curso);
  printf("Promedio Ponderado %f\n",Promedio);
}

void Asistente::Captura(void) {
  Persona::Captura();
  printf("Carrera:           ");
  flushall();
```

```
        gets(Carrera);
  printf("Horas Asignadas:   ");
  scanf("%d",&HorasAs);
  printf("Curso Asignado:    ");
  flushall();
  gets(Curso);
  printf("Promedio Ponderado ");
  scanf("%f",&Promedio);
}

float Asistente::ObtPromedio(void) {
  return Promedio;
}

int Asistente::ObtHorasAs(void) {
  return HorasAs;
}

void Asistente::ObtCurso(char Cur[50]) {
  strcpy(Cur,Curso);
}

void Asistente::ObtCarrera(char Carr[50]) {
  strcpy(Carr,Carrera);
}

void Asistente::CambiaPromedio(float NuevoProm) {
  Promedio = NuevoProm;
}

void Asistente::CambiaHorasAs(int NuevaHo) {
  HorasAs = NuevaHo;
}

void Asistente::CambiaCarrera(char NuevaCarrera[50]) {
  strcpy(Carrera,NuevaCarrera);
}

void Asistente::CambiaCurso(char NuevoCurso[50]) {
  strcpy(Curso,NuevoCurso);
}
```

```
void Asistente::Guardar(FILE *archivo) {
  char aux='A';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
  fwrite(&HorasAs,sizeof(int),1,archivo);
  fwrite(Curso,MAX2*sizeof(char),1,archivo);
  fwrite(Carrera,MAX2*sizeof(char),1,archivo);
}

void Asistente::Recuperar(FILE *archivo) {
  Persona::Recuperar(archivo);
  fread(&Promedio,sizeof(float),1,archivo);
  fread(&HorasAs,sizeof(int),1,archivo);
  fread(Curso,MAX2*sizeof(char),1,archivo);
  fread(Carrera,MAX2*sizeof(char),1,archivo);
}

// LPERS26.HPP
// protege para múltiple inclusión
#if ! defined(_I_LPERS26_HPP_)
#define _I_LPERS26_HPP_
#include "pers26.hpp"

struct Nodo {
   Persona *PPersona;
   Nodo    *Sig;
};

class ListaPersonas {
  Nodo *Primera;
public:
  ListaPersonas();
  ~ListaPersonas();
  void Inserta(Persona *NuevaPersona);
  void Muestra(void);
  void Guardar(FILE *archivo);
  void Recuperar(FILE *archivo);
};
```

```
  #endif
```

// LPERS26.CPP

```
#include <stdio.h>
#include "lpers26.hpp"
#include "pers26.hpp"

ListaPersonas::ListaPersonas() {
  Primera = NULL;
}

ListaPersonas::~ListaPersonas() {
  while(Primera != NULL) {
    Nodo *Tempo=Primera;
    Primera=Primera->Sig;
    delete Tempo->PPersona;
    delete Tempo;
  }
}

void ListaPersonas::Inserta(Persona *NuevaPersona) {
  Nodo *NodoTemp, *NuevoNodo;
  NuevoNodo = new Nodo;
  NuevoNodo->PPersona = NuevaPersona;
  NuevoNodo->Sig = NULL;
  if(Primera==NULL)
    Primera = NuevoNodo;
  else {
     NodoTemp=Primera;
     while(NodoTemp->Sig != NULL)
       NodoTemp=NodoTemp->Sig;
        NodoTemp->Sig=NuevoNodo;
  }
}

void ListaPersonas::Muestra() {
 Nodo *Actual=Primera;
 while(Actual != NULL) {
     Actual->PPersona->Muestra();
     printf("\n");
     Actual = Actual->Sig;
```

```
      }
}

void ListaPersonas::Guardar(FILE *archivo)  {
  Nodo *Actual=Primera;
  archivo = fopen("personas.dat","w");
  while(Actual != NULL) {
      Actual->PPersona->Guardar(archivo);
      Actual = Actual->Sig;
  }
  fclose(archivo);
}

void ListaPersonas::Recuperar(FILE *archivo)  {
  char clave;
  ListaPersonas::~ListaPersonas();
  archivo = fopen("personas.dat","r");
  while(!feof(archivo)) {
    if(fread(&clave,sizeof(char),1,archivo)!=0) {
      switch(clave) {
        case 'P': Persona *PPersona=new Persona();
                  PPersona->Recuperar(archivo);
                    Inserta(PPersona);
                   break;
        case 'E': Estudiante *PEstudiante=new Estudiante();
                  PEstudiante->Recuperar(archivo);
                    Inserta(PEstudiante);
                    break;
        case 'C': EstCompu *PEstCompu=new EstCompu();
                  PEstCompu->Recuperar(archivo);
                    Inserta(PEstCompu);
                    break;
        case 'A': Asistente *PAsistente=new Asistente();
                  PAsistente->Recuperar(archivo);
                    Inserta(PAsistente);
                    break;
      }
    }
  }
  fclose(archivo);
}
```

// E26.CPP

```
#include "pers26.hpp"
#include "lpers26.hpp"
#include <conio.h>
#include <stdio.h>

void LeeDatosPersona(void);
void LeeDatosAsistente(void);
void LeeDatosEstudiante(void);
void LeeDatosEstCompu(void);
char Menu(void);
Asistente *PAsistente;
Persona *PPersona;
Estudiante *PEstudiante;
EstCompu *PEstCompu;

void main() {
  ListaPersonas UnaLista;
  FILE *Arch;
  char opcion;
  do {
    clrscr();
    opcion = Menu();
    switch(opcion) {
      case '1': clrscr();
              LeeDatosPersona();
                 UnaLista.Inserta(PPersona);
                 break;
      case '2': clrscr();
                 LeeDatosAsistente();
                 UnaLista.Inserta(PAsistente);
                 break;
      case '3': clrscr();
                 LeeDatosEstudiante();
                 UnaLista.Inserta(PEstudiante);
                 break;
      case '4': clrscr();
                 LeeDatosEstCompu();
                 UnaLista.Inserta(PEstCompu);
                 break;
```

```
        case '5': clrscr();
                  UnaLista.Muestra();
                  getch();
                  break;
     case '6': clrscr();
                  UnaLista.Guardar(Arch);
                  printf("Lista guardada satisfactoriamente");
                  getch();
                  break;
     case '7': clrscr();
                  UnaLista.Recuperar(Arch);
                  printf("Lista recuperada satisfactoriamente");
                  getch();
                  break;
   }
 } while(opcion!='8');
}

void LeeDatosPersona() {
 PPersona=new Persona();
 PPersona->Captura();
}

void LeeDatosEstudiante() {
 PEstudiante=new Estudiante();
 PEstudiante->Captura();
}

void LeeDatosEstCompu() {
 PEstCompu=new EstCompu();
 PEstCompu->Captura();
}

void LeeDatosAsistente() {
 PAsistente = new Asistente();
 PAsistente->Captura();
}

char Menu(void) {
   char opcion;
   printf("[1] Insertar Persona\n");
```

```
      printf("[2] Insertar Asistente\n");
   printf("[3] Insertar Estudiante\n");
   printf("[4] Insertar Estudiante de Computación\n");
   printf("[5] Ver la Lista Polimórfica\n");
   printf("[6] Guardar la Lista Polimórfica\n");
   printf("[7] Recuperar la Lista Polimórfica\n");
   printf("[8] Salir\n");
   opcion= getch();
   return opcion;
}
```

Para iniciar el análisis de este ejemplo nótese en el siguiente fragmento de código que el método Guardar de la clase ListaPersonas tiene un ciclo de llamadas a los métodos Guardar de las clases Persona, Estudiante, EstCompu y Asistente, estos llamados son determinados en tiempo de ejecución ("Binding Tardío") ya que los métodos Guardar de las clases Persona, Estudiante, EstCompu y Asistente son todos virtuales y el llamado es hecho a través del puntero PPersona que es un puntero a la clase Persona cabeza de la jerarquía.

```
void ListaPersonas::Guardar(FILE *archivo)  {
  Nodo *Actual=Primera;
  archivo = fopen("personas.dat","w");
  while(Actual != NULL) {
     Actual->PPersona->Guardar(archivo);      // Binding Tardío
     Actual = Actual->Sig;
  }
  fclose(archivo);
}
```

Ahora observe el método Guardar de la clase Persona, note que este método permite almacenar en disco una clave, en este caso la letra 'P', que será necesaria cuando se recupere el archivo y además almacena en disco todos los atributos de una Persona, a saber Edad y Nombre.

```
void Persona::Guardar(FILE *archivo) {
  char aux='P';
  fwrite(&aux,sizeof(char),1,archivo);
```

```
   fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
}
```

De manera similar funciona el método Guardar de las demás clases. Sin embargo los métodos Guardar de las clases Estudiante y EstCompu tienen una característica especial y es que deben almacenar en disco los atributos provenientes de la relación Com-Com con la clase Libro, esto se logra debido a que en el método Guardar de la clase Estudiante invoca al método Guardar de la clase Libro, como se muestra a continuación:

```
void Estudiante::Guardar(FILE *archivo) {
  char aux='E';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Examen1,sizeof(float),1,archivo);
  fwrite(&Examen2,sizeof(float),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
  L.Guardar(archivo);                          // Relación Com-Com
}
```

donde el método Guardar de la clase Libro tiene el siguiente código:

```
void Libro::Guardar(FILE *archivo) {
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Anno,sizeof(int),1,archivo);
}
```

nótese que en este método no se almacena ningún tipo de clave, esto por cuanto Libro no es por sí mismo un nodo de la lista, sino que solamente es un componente de la clase Estudiante y EstuCompu.

Los métodos de recuperación de disco no funcionan de una forma tan elegante como los métodos de almacenamiento, porque cuando se lee un registro del disco C++ no puede, al menos con archivos tipo C, de determinar en forma automática qué tipo de registro está recuperando. Debido a lo anterior, se hace necesario almacenar una clave, la cual

es leída antes de recuperar el registro, para determinar el "tipo" de registro que será leído, como se muestra a continuación en el método Recuperar de la clase ListaPersonas:

```
void ListaPersonas::Recuperar(FILE *archivo)  {
  char clave;
  ListaPersonas::~ListaPersonas();
  archivo = fopen("personas.dat","r");
  while(!feof(archivo)) {
    if(fread(&clave,sizeof(char),1,archivo)!=0) {
      switch(clave) {
        case 'P': Persona *PPersona=new Persona();
                  PPersona->Recuperar(archivo);
                    Inserta(PPersona);
                   break;
        case 'E': Estudiante *PEstudiante=new Estudiante();
                  PEstudiante->Recuperar(archivo);
                    Inserta(PEstudiante);
                    break;
        case 'C': EstCompu *PEstCompu=new EstCompu();
                  PEstCompu->Recuperar(archivo);
                    Inserta(PEstCompu);
                    break;
        case 'A': Asistente *PAsistente=new Asistente();
                  PAsistente->Recuperar(archivo);
                    Inserta(PAsistente);
                    break;
      }
    }
  }
  fclose(archivo);
}
```

Los métodos Recuperar de las clases Persona, Estudiante, EstCompu y Asistente se encargan de leer del disco los atributos correspondientes a cada clase, por ejemplo el código del método Recuperar de la clase Estudiante es el siguiente:

```
void Estudiante::Recuperar(FILE *archivo) {
```

```
    Persona::Recuperar(archivo);
 fread(&Examen1,sizeof(float),1,archivo);
 fread(&Examen2,sizeof(float),1,archivo);
 fread(&Promedio,sizeof(float),1,archivo);
 L.Recuperar(archivo);                        // Relación Com-Com
}
```

observe que los atributos de la clase Libro se recuperan a través de la invocación del método Recuperar de la clase Libro.

2.6 Listas genéricas

En la sección anterior se implementó la relación Com-Com 1-n entre la clase Estudiante y la clase Libro mediante un arreglo, pero esto

genera limitaciones ya conocidas en cuanto al número máximo de libros que podrá tener un estudiante, además en otras ocasiones provoca desperdicios de memoria debido a que no se usa todo el arreglo.

La implementación óptima de una relación Com-Com 1-n es mediante una lista enlazada, en este caso una lista enlazada de libros, tal como se ilustra en la Figura 2.10. Pero la pregunta es cómo aprovechar el código de la lista de Personas que ya se tiene implementada? Es decir, cómo aprovechar la Orientación a Objetos para no programar dos listas enlazadas cuyos algoritmos son idénticos (la diferencia está el tipo del nodo)? La Orientación a Objetos proporciona dos formas: la primera mediante listas genéricas y la segunda mediante las listas (clases) parametrizadas. En esta sección presentamos la primera solución y en la sección 2.7 presentamos la segunda solución.

FIGURA 2.10. Relación 1-n mediante una lista

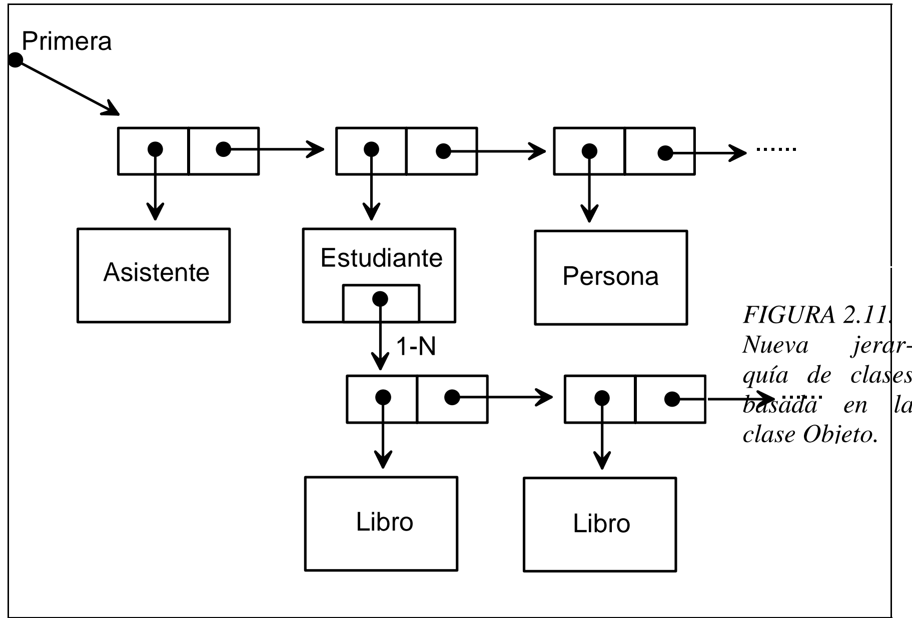
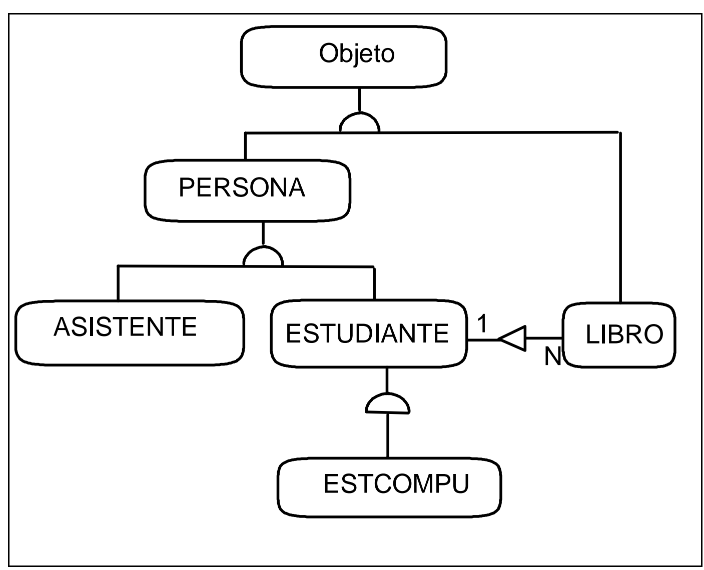


FIGURA 2.11. Nueva jerarquía de clases basada en la clase Objeto.

A continuación presentamos nuevamente el ejemplo 2.6, pero en esta nueva versión se introduce una nueva clase denominada ***Objeto*** de la cual se derivarán las clases Persona, Estudiante, Asistente, EstCompu y Libro para luego implementar una Lista de Objetos, en lugar de una lista de Personas como se hizo en el ejemplo 2.6. Esta nueva jerarquía de clases se presenta en la Figura 2.11.



🖫 Ejemplo 2.7: Polimorfismo mediante una lista genérica

```
// OBJETO.HPP
#if ! defined(_I_OBJETO_HPP_)
#define _I_OBJETO_HPP_

class Objeto {
public:
  Objeto() { };
        virtual void Muestra(void) { };
        virtual void Captura(void) { };
        virtual void Guardar(FILE *archivo) { };
        virtual void Recuperar(FILE *archivo){ };
};
#endif

// PERS27.HPP
// protege para múltiple inclusión
#if ! defined(_I_PERS27_HPP_)
#define _I_PERS27_HPP_

#include <stdio.h>
#include "objeto.hpp"
```

```
    #include "lpers27.hpp"

#define MAX1 30
#define MAX2 50

class Libro : public Objeto {
        char Nombre[30];
        int Anno;
public:
        Libro();
        virtual void Muestra(void);
        virtual void Captura(void);
        virtual void CambiaNombre(char NuevoNom[30]);
        virtual void CambiaAnno(int NuevoAnno);
        virtual void ObtNombre(char Nom[30]);
        virtual int  ObtAnno(void);
        virtual void Guardar(FILE *archivo);
        virtual void Recuperar(FILE *archivo);
};

class Persona : public Objeto {
protected:              // Permite que estos atributos sean
        char Nombre[30];        // conocidos en las clases derivadas
        int     Edad;
public:
        Persona(char Nom[30], int Ed);
        Persona();
        virtual void Muestra(void);
        virtual void Captura(void);
        virtual void CambiaNombre(char NuevoNom[30]);
        virtual void CambiaEdad(int NuevaEdad);
  virtual void ObtNombre(char Nom[30]);
  virtual int  ObtEdad(void);
  virtual void Guardar(FILE *archivo);
  virtual void Recuperar(FILE *archivo);
};

class Estudiante : public Persona {
protected:
  float Examen1;
  float Examen2;
```

```
        float Promedio;
    int NumeroLibros;
    ListaObjetos L;                // Relación Com-Com
public:
    Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
    Estudiante();
    virtual void  CalculaPromedio(void);
    virtual void  Muestra(void);
    virtual void  Captura(void);
    void  CambiaNotas(float NuevaNota1, float NuevaNota2);
    virtual float ObtEx1(void);
    virtual float ObtEx2(void);
    virtual float ObtProm(void);
    virtual void Guardar(FILE *archivo);
    virtual void Recuperar(FILE *archivo);
};

class EstCompu : public Estudiante {
    float  Examen3;
public:
    EstCompu(char Nom[30],int Ed, float Ex1, float Ex2, float Ex3);
    EstCompu();
    // No requiere método Muestra pues lo hereda sin ningún cambio
    virtual void Captura(void);
    void CambiaNotas(float NuevaNota1, float NuevaNota2, float Nueva-
Nota3);
    virtual void CalculaPromedio(void);
    virtual float ObtEx3(void);
    virtual void Guardar(FILE *archivo);
    virtual void Recuperar(FILE *archivo);
};

class Asistente : public Persona {
  float Promedio;
  int HorasAs;
  char Curso[50];
  char Carrera[50];
public:
  Asistente(char Nom[30],int Ed,float Prom,char Cur[50],int HA,char Carr[50]);
  Asistente();
  virtual void Muestra(void);
```

```
        virtual void Captura(void);
   virtual float ObtPromedio(void);
   virtual int  ObtHorasAs(void);
   virtual void ObtCurso(char Cur[50]);
   virtual void ObtCarrera(char Carr[50]);
   virtual void CambiaPromedio(float NuevoProm);
   virtual void CambiaHorasAs(int NuevaHo);
   virtual void CambiaCurso(char NuevoCurso[50]);
   virtual void CambiaCarrera(char NuevaCarrera[50]);
   virtual void Guardar(FILE *archivo);
   virtual void Recuperar(FILE *archivo);
};
#endif
```

// PERS27.CPP

```
#include "pers27.hpp"
#include <string.h>
#include <stdio.h>

/*-----------------------P E R S O N A-----------------------------*/

Persona::Persona(char Nom[30],int Ed) {
  strcpy(Nombre,Nom);
  Edad = Ed;
}

Persona::Persona() {
  strcpy(Nombre,"");
  Edad = 0;
}

void Persona::Muestra(void) {
  printf("Persona: %s\n",Nombre);
  printf("Edad:    %d\n",Edad);
}

void  Persona::Captura(void){
  printf("Nombre:    ");
  flushall();
  gets(Nombre);
  printf("Edad: ");
```

```
      scanf("%d",&Edad);
}

void Persona::CambiaNombre(char NuevoNom[30]) {
  strcpy(Nombre,NuevoNom);
}

void Persona::CambiaEdad(int NuevaEdad) {
  Edad = NuevaEdad;
}

void Persona::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}

int Persona::ObtEdad(void) {
  return Edad;
}

void Persona::Guardar(FILE *archivo) {
  char aux='P';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
}

void Persona::Recuperar(FILE *archivo) {
  fread(&Edad,sizeof(int),1,archivo);
  fread(Nombre,MAX1*sizeof(char),1,archivo);
}

/*-----------------------L I B R O-----------------------------*/

Libro::Libro() {
  strcpy(Nombre,"");
  Anno = 0;
}

void Libro::Muestra(void) {
  printf("Nombre del libro: %s\n",Nombre);
  printf("Año del libro:    %d\n",Anno);
```

```
    }

void Libro::Captura(void) {
  printf("Nombre del Libro: ");
  flushall();
  gets(Nombre);
  printf("Año del Libro:    ");
  scanf("%d",&Anno);
}

void Libro::CambiaNombre(char NuevoNom[30]) {
  strcpy(Nombre,NuevoNom);
}

void Libro::CambiaAnno(int NuevoAnno) {
  Anno = NuevoAnno;
}

void Libro::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}

int Libro::ObtAnno(void) {
  return Anno;
}

void Libro::Guardar(FILE *archivo) {
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Anno,sizeof(int),1,archivo);
}

void Libro::Recuperar(FILE *archivo) {
  fread(Nombre,MAX1*sizeof(char),1,archivo);
  fread(&Anno,sizeof(int),1,archivo);
}

/*------------------------E S T U D I A N T E---------------------------*/

// Llama al constructor de la clase base
Estudiante::Estudiante(char Nom[30],int Ed,float Ex1,float Ex2) :
                                                    Persona(Nom,Ed) {
```

```
       Examen1 = Ex1;
  Examen2 = Ex2;
}

Estudiante::Estudiante() : Persona() {
  Examen1 = 0;
  Examen2 = 0;
}

void Estudiante::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2)/2;
}

void Estudiante::Muestra(void)  {
  printf("Estudiante: %s\n",Nombre);
  printf("Promedio  : %f\n",Promedio);
  L.Muestra();  // Se encarga de mostrar el Libro
}

// Programación Incremental
void Estudiante::Captura(void)  {
  Persona::Captura();                  // Invoca a Captura de la clase base
  printf("Nota Examen 1: ");
  scanf("%f",&Examen1);
  printf("Nota Examen 2: ");
  scanf("%f",&Examen2);
  Promedio=(Examen1+Examen2)/2;
  printf("Número de libros del estudiante: ");
  scanf("%d",&NumeroLibros);
  Libro *PLibro;
  for(int i=1; i<=NumeroLibros; i++) {
                        PLibro=new Libro();
                        PLibro->Captura();
                        L.Inserta(PLibro);
  }
}

void Estudiante::CambiaNotas(float NuevaNota1, float NuevaNota2) {
  Examen1 = NuevaNota1;
  Examen2 = NuevaNota2;
}
```

```
float Estudiante::ObtEx1(void) {
  return Examen1;
}

float Estudiante::ObtEx2(void) {
  return Examen1;
}

float Estudiante::ObtProm(void) {
  return Promedio;
}

void Estudiante::Guardar(FILE *archivo) {
  char aux='E';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Examen1,sizeof(float),1,archivo);
  fwrite(&Examen2,sizeof(float),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
  L.Guardar(archivo,NumeroLibros);                // Relación Com-Com
}

void Estudiante::Recuperar(FILE *archivo) {
  Persona::Recuperar(archivo);
  fread(&Examen1,sizeof(float),1,archivo);
  fread(&Examen2,sizeof(float),1,archivo);
  fread(&Promedio,sizeof(float),1,archivo);
  L.Recuperar(archivo,NumeroLibros);                     // Relación Com-Com
}

/*------------------------E S T - C O M P---------------------------*/

// Llama al constructor de la clase base
EstCompu::EstCompu(char Nom[30],int Ed,float Ex1,float Ex2,float Ex3) :                    Estudiante
        Examen3 = Ex3;
}

void EstCompu::Captura(void)  {
  Persona::Captura();               // Invoca a Captura de la clase Persona
```

```
      printf("Nota Examen 1: ");        // No invoca a Captura de la clase base
  scanf("%f",&Examen1);            // pues debe capturar 3 exámenes y no 2
  printf("Nota Examen 2: ");       // y porque la forma de calcular el promedio
  scanf("%f",&Examen2);            // es diferente
  printf("Nota Examen 3: ");
  scanf("%f",&Examen3);
  Promedio=(Examen1+Examen2+Examen3)/3;
  printf("Número de libros del estudiante: ");
  scanf("%d",&NumeroLibros);
  Libro *PLibro;
  for(int i=1; i<=NumeroLibros; i++) {
                PLibro=new Libro();
                PLibro->Captura();
                L.Inserta(PLibro);
  }
}

EstCompu::EstCompu() : Estudiante() {
        Examen3 = 0;
}

void EstCompu::CambiaNotas(float NuevaNota1, float NuevaNota2,
                                              float NuevaNota3) {
  Examen1 = NuevaNota1;
  Examen2 = NuevaNota2;
  Examen3 = NuevaNota3;
}

void EstCompu::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2 + Examen3)/3;
}

float EstCompu::ObtEx3(void) {
  return Examen3;
}

void EstCompu::Guardar(FILE *archivo) {
  char aux='C';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
```

```
        fwrite(&Examen1,sizeof(float),1,archivo);
  fwrite(&Examen2,sizeof(float),1,archivo);
  fwrite(&Examen3,sizeof(float),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
  L.Guardar(archivo,NumeroLibros);
}

void EstCompu::Recuperar(FILE *archivo) {
  Persona::Recuperar(archivo);
  fread(&Examen1,sizeof(float),1,archivo);
  fread(&Examen2,sizeof(float),1,archivo);
  fread(&Examen3,sizeof(float),1,archivo);
  fread(&Promedio,sizeof(float),1,archivo);
  L.Recuperar(archivo,NumeroLibros);
}

/*-----------------------A S I S T E N T E----------------------------*/

Asistente::Asistente(char Nom[30],int Ed,float Prom,char Cur[50],
                                   int HA,char Carr[50]) : Persona(Nom,Ed) {
   Promedio = Prom;
   strcpy(Curso,Cur);
   HorasAs = HA;
   strcpy(Carrera,Carr);
}

Asistente::Asistente(){
   Promedio = 0;
   strcpy(Curso,"");
   HorasAs = 0;
   strcpy(Carrera,"");
}

void Asistente::Muestra(void) {
  printf("Asistente:         %s\n",Nombre);
  printf("Edad:              %d\n",Edad);
  printf("Carrera:           %s\n",Carrera);
  printf("Horas Asignadas:   %d\n",HorasAs);
  printf("Curso Asignado:    %s\n",Curso);
  printf("Promedio Ponderado %f\n",Promedio);
}
```

```
void Asistente::Captura(void) {
  Persona::Captura();
  printf("Carrera:           ");
  flushall();
  gets(Carrera);
  printf("Horas Asignadas:   ");
  scanf("%d",&HorasAs);
  printf("Curso Asignado:    ");
  flushall();
  gets(Curso);
  printf("Promedio Ponderado ");
  scanf("%f",&Promedio);
}

float Asistente::ObtPromedio(void) {
  return Promedio;
}

int Asistente::ObtHorasAs(void) {
  return HorasAs;
}

void Asistente::ObtCurso(char Cur[50]) {
  strcpy(Cur,Curso);
}

void Asistente::ObtCarrera(char Carr[50]) {
  strcpy(Carr,Carrera);
}

void Asistente::CambiaPromedio(float NuevoProm) {
  Promedio = NuevoProm;
}

void Asistente::CambiaHorasAs(int NuevaHo) {
  HorasAs = NuevaHo;
}

void Asistente::CambiaCarrera(char NuevaCarrera[50]) {
  strcpy(Carrera,NuevaCarrera);
```

```
    }

void Asistente::CambiaCurso(char NuevoCurso[50]) {
  strcpy(Curso,NuevoCurso);
}

void Asistente::Guardar(FILE *archivo) {
  char aux='A';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
  fwrite(&HorasAs,sizeof(int),1,archivo);
  fwrite(Curso,MAX2*sizeof(char),1,archivo);
  fwrite(Carrera,MAX2*sizeof(char),1,archivo);
}

void Asistente::Recuperar(FILE *archivo) {
  Persona::Recuperar(archivo);
  fread(&Promedio,sizeof(float),1,archivo);
  fread(&HorasAs,sizeof(int),1,archivo);
  fread(Curso,MAX2*sizeof(char),1,archivo);
  fread(Carrera,MAX2*sizeof(char),1,archivo);
}

// LPERS27.HPP
// protege para múltiple inclusión
#if ! defined(_I_LPERS27_HPP_)
#define _I_LPERS27_HPP_

#include "objeto.hpp"

struct Nodo {
        Objeto *PObjeto;
        Nodo   *Sig;
};

class ListaObjetos {
  Nodo *Primera;
public:
  ListaObjetos();
```

```
        ~ListaObjetos();
  void Inserta(Objeto *NuevoObjeto);
  void Muestra(void);
  void Guardar(FILE *archivo);
  void Recuperar(FILE *archivo);
  void Guardar(FILE *archivo, int N);
  void Recuperar(FILE *archivo, int &N);
};
#endif

// LPERS27.CPP
#include <stdio.h>
#include "lpers27.hpp"
#include "pers27.hpp"

ListaObjetos::ListaObjetos() {
        Primera = NULL;
}

ListaObjetos::~ListaObjetos() {
   while(Primera != NULL) {
      Nodo *Tempo=Primera;
      Primera=Primera->Sig;
      delete Tempo->PObjeto;
                delete Tempo;
   }
}

void ListaObjetos::Inserta(Objeto *NuevoObjeto) {
        Nodo *NodoTemp, *NuevoNodo;
        NuevoNodo = new Nodo;
        NuevoNodo->PObjeto = NuevoObjeto;
        NuevoNodo->Sig = NULL;
        if(Primera==NULL)
                Primera = NuevoNodo;
        else {
                 NodoTemp=Primera;
                 while(NodoTemp->Sig != NULL)
                         NodoTemp=NodoTemp->Sig;
                 NodoTemp->Sig=NuevoNodo;
        }
```

```
    }

void ListaObjetos::Muestra() {
  Nodo *Actual=Primera;
  while(Actual != NULL) {
                Actual->PObjeto->Muestra();
                printf("\n");
                Actual = Actual->Sig;
  }
}

void ListaObjetos::Guardar(FILE *archivo)  {
  Nodo *Actual=Primera;
  archivo = fopen("personas.dat","w");
  while(Actual != NULL) {
                Actual->PObjeto->Guardar(archivo);
                Actual = Actual->Sig;
  }
  fclose(archivo);
}

void ListaObjetos::Recuperar(FILE *archivo)  {
  char clave;
  ListaObjetos::~ListaObjetos();
  archivo = fopen("personas.dat","r");
  while(!feof(archivo)) {
        if(fread(&clave,sizeof(char),1,archivo)!=0) {
                switch(clave) {
                  case 'P': Persona *PObjeto=new Persona();
                            PObjeto->Recuperar(archivo);
                            Inserta(PObjeto);
                            break;
                  case 'E': Estudiante *PEstudiante=new Estudiante();
                            PEstudiante->Recuperar(archivo);
                            Inserta(PEstudiante);
                            break;
                  case 'C': EstCompu *PEstCompu=new EstCompu();
                            PEstCompu->Recuperar(archivo);
                            Inserta(PEstCompu);
                            break;
                  case 'A': Asistente *PAsistente=new Asistente();
```

```
                                PAsistente->Recuperar(archivo);
                             Inserta(PAsistente);
                             break;
                    }
            }
  }
  fclose(archivo);
}

void ListaObjetos::Recuperar(FILE *archivo,int &N)  {
  if(fread(&N,sizeof(int),1,archivo)!=0) {
            for(int i=1; i<=N; i++) {
                            Libro *PLibro=new Libro();
                            PLibro->Recuperar(archivo);
                            Inserta(PLibro);
            }
  }
}

void ListaObjetos::Guardar(FILE *archivo,int N)  {
  Nodo *Actual=Primera;
  if(fwrite(&N,sizeof(int),1,archivo)!=0) {
            while(Actual != NULL) {
                      Actual->PObjeto->Guardar(archivo);
                      Actual = Actual->Sig;
            }
  }
}

// Programa de Prueba
// E27.CPP
#include "pers27.hpp"
#include "lpers27.hpp"
#include <conio.h>
#include <stdio.h>

void LeeDatosPersona(void);
void LeeDatosAsistente(void);
void LeeDatosEstudiante(void);
void LeeDatosEstCompu(void);
char Menu(void);
```

```
      Asistente *PAsistente;
Persona *PPersona;
Estudiante *PEstudiante;
EstCompu *PEstCompu;
FILE *Arch;

void main() {
  ListaObjetos UnaLista;
  char opcion;
  do {
             clrscr();
              opcion = Menu();
     switch(opcion) {
                          case '1': clrscr();
                                    LeeDatosPersona();
                                    UnaLista.Inserta(PPersona);
                                    break;
                          case '2': clrscr();
                                    LeeDatosAsistente();
                                    UnaLista.Inserta(PAsistente);
                                    break;
                          case '3': clrscr();
                                    LeeDatosEstudiante();
                                    UnaLista.Inserta(PEstudiante);
                                    break;
                          case '4': clrscr();
                                    LeeDatosEstCompu();
                                    UnaLista.Inserta(PEstCompu);
                                    break;
                          case '5': clrscr();
                                    UnaLista.Muestra();
                                    getch();
                                    break;
                          case '6': clrscr();
                                    UnaLista.Guardar(Arch);
                                    printf("Lista guardada satisfactoriamente");
                                    getch();
                                    break;
                          case '7': clrscr();
                                    UnaLista.Recuperar(Arch);
                                    printf("Lista recuperada                                    Satisfactori
```

```
                                    getch();
                               break;
            }
  } while(opcion!='8');
}

void LeeDatosPersona() {
  PPersona=new Persona();
  PPersona->Captura();
}

void LeeDatosEstudiante() {
  PEstudiante=new Estudiante();
  PEstudiante->Captura();
}

void LeeDatosEstCompu() {
  PEstCompu=new EstCompu();
  PEstCompu->Captura();
}

void LeeDatosAsistente() {
  PAsistente = new Asistente();
  PAsistente->Captura();
}

char Menu(void) {
    char opcion;
    printf("[1] Insertar Persona\n");
    printf("[2] Insertar Asistente\n");
    printf("[3] Insertar Estudiante\n");
    printf("[4] Insertar Estudiante de Computación\n");
    printf("[5] Ver la Lista Polimórfica\n");
    printf("[6] Guardar la Lista Polimórfica\n");
    printf("[7] Recuperar la Lista Polimórfica\n");
    printf("[8] Salir\n");
    opcion= getch();
    return opcion;
}
```

La idea central de las listas genéricas es crear un tipo Objeto mediante el cual se agrupen vía herencia todas las clases. La clase Objeto debe estar en un archivo independiente, para que pueda ser reutilizable en cualquier otra jerarquía de clases, así como para evitar problemas de "include" circulares. Esta clase Objeto no tiene código, pero en se deben definir todos los métodos virtuales que sean necesarios en las clases derivadas para implementar la lista genérica. Tales métodos tendrán código "inline" vacío, como puede verse en el siguiente fragmento de código:

```
class Objeto {
public:
  Objeto() { };
        virtual void Muestra(void) { };
        virtual void Captura(void) { };
        virtual void Guardar(FILE *archivo) { };
        virtual void Recuperar(FILE *archivo){ };
};
```

Ahora bien, en este ejemplo se implementa una lista de Objetos, de modo tal que para implementar una lista enlazada de algún "tipo" clase, bastará definir esta clase como derivada de la clase Objeto. La definición de esta lista genérica de objetos se presenta en el siguiente fragmento de código:

```
struct Nodo {
        Objeto *PObjeto;
        Nodo   *Sig;
};

class ListaObjetos {
  Nodo *Primera;
public:
  ListaObjetos();
  ~ListaObjetos();
  void Inserta(Objeto *NuevoObjeto);
  void Muestra(void);
  void Guardar(FILE *archivo);
```

```
    void Recuperar(FILE *archivo);
  void Guardar(FILE *archivo, int N);
  void Recuperar(FILE *archivo, int &N);
};
```

El código y los algoritmos de esta lista no difieren en nada respecto a la implementación del ejemplo 2.6, excepto que en lugar de punteros a Persona se tienen punteros a Objeto.

Dado que en este ejemplo se desea una lista de Personas y otra lista Libros basta definir estas clases como derivadas de la clase Objeto, tal como se muestra en el siguiente fragmento de código:

```
class Libro : public Objeto {
        char Nombre[30];
        int Anno;
public:
        Libro();
        virtual void Muestra(void);
        virtual void Captura(void);
        virtual void CambiaNombre(char NuevoNom[30]);
        virtual void CambiaAnno(int NuevoAnno);
        virtual void ObtNombre(char Nom[30]);
        virtual int  ObtAnno(void);
        virtual void Guardar(FILE *archivo);
        virtual void Recuperar(FILE *archivo);
};

class Persona : public Objeto {
protected:                  // Permite que estos atributos sean
        char Nombre[30];        // conocidos en las clases derivadas
        int     Edad;
public:
        Persona(char Nom[30], int Ed);
        Persona();
        virtual void Muestra(void);
        virtual void Captura(void);
        virtual void CambiaNombre(char NuevoNom[30]);
        virtual void CambiaEdad(int NuevaEdad);
```

```
      virtual void ObtNombre(char Nom[30]);
      virtual int  ObtEdad(void);
      virtual void Guardar(FILE *archivo);
      virtual void Recuperar(FILE *archivo);
};
```

De esta manera la relación Com-Com entre Libro y Estudiante se puede implementar mediante una lista enlazada. Nótese que en este ejemplo se tiene una lista enlazada, algunos de cuyos nodos (los de tipo Estudiante o EstCompu) tendrán a su vez una lista enlazada, aprovechando el mismo código para ambas listas. Esto se puede ver en la definición de la clase Estudiante, que se presenta seguidamente:

```
class Estudiante : public Persona {
protected:
  float Examen1;
  float Examen2;
      float Promedio;
      int NumeroLibros;
      ListaObjetos L;                    // Relación Com-Com
public:
      Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
      . . . . . . . .
      virtual void Recuperar(FILE *archivo);
};
```

Para Capturar, Mostrar, Insertar los Libros a un Estudiante se usan los métodos de la Lista a través de la instancia L. Fue necesario en este ejemplo implementar nuevos métodos Guardar y Recuperar a la lista genérica, esto con el fin de grabar y recuperar de disco los Libros del Estudiante, debido a que en este caso se conoce a priori el número de Libros que tendrá un Estudiante y debido fundamentalmente a que los Libros de un Estudiante se graban en el mismo archivo en que se graba la lista polimórfica (por lo que en este caso los métodos Guardar y Recuperar no deben abrir nuevamente el archivo, pues lo destruirían).

En la siguiente sección se presentará una nueva versión del ejemplo 2.7 utilizando la idea de tipos o clases parametrizadas.

2.7 Tipos (clases) parametrizados (Templates)

La noción de "templates" provee los llamados algunas veces Tipos Parametrizados o Genéricos. Para esto se reincorporan en C++ conceptos que estaban presentes en lenguajes como Clu y Ada. [Stroustrup-2]

Es muy común en programación implementar una misma estructura de datos varias veces debido a que el tipo del nodo es diferente. Por ejemplo, implementar una lista de Personas y luego implementar una

lista de Libros ambas con los mismos algoritmos. Para evitar esta redundancia de código, *Bjarne Stroustrup* introduce en C++ la idea de que una clase pueda recibir un tipo o varios tipos como parámetro. De este modo, tal como se ilustra en el ejemplo 2.10, se puede utilizar la misma implementación de una lista para Personas y Libros o incluso para cualquier nueva clase que en el futuro sea implementada.

Antes de presentar la nueva implementación de la lista del ejemplo 2.7, dedicaremos esta sección a explicar con detalle la sintaxis y la semántica de los tipos parametrizados. Para ilustrar este nuevo concepto implementaremos en el ejemplo 2.8 una Pila Parametrizada mediante un arreglo.

🖫 Ejemplo 2.8: Implementación de una pila parametrizada

```
// E28.CPP
#include <iostream.h>
#include "pers26.hpp"
#define VACIO -1
#define MAX 100

template <class TIPO>
class Pila {
	TIPO* Datos;
	int Top;
public:
	Pila() {
		Datos = new TIPO[MAX];
		Top = VACIO;
```

```
        }
    ~Pila() {delete [] Datos; }
    void Insertar(TIPO c);
    TIPO Sacar();
    int Vacia();
    int Llena();
};

template <class TIPO>
void Pila<TIPO>::Insertar(TIPO c)  {
  Datos[++Top] = c;
}

template <class TIPO>
TIPO Pila<TIPO>::Sacar() {
    return (Datos[Top--]);
}

template <class TIPO>
int Pila<TIPO>::Vacia() {
    return (Top==VACIO);
}

template <class TIPO>
int Pila<TIPO>::Llena() {
    return (Top==MAX-1);
}

main() {
  Pila<int> x;
  x.Insertar(1);
  x.Insertar(2);
  x.Insertar(3);
  while(!x.Vacia())
      cout << x.Sacar() << "\n";
  cout << "Termino" << "\n\n";
  Pila<Persona*> P;
  Persona *P1=new Persona();
  Persona *P2=new Persona();
  Persona *P3=new Persona();
  Estudiante *E1 = new Estudiante();
```

```
      Persona *Aux = new Persona();
  P1->Captura();
  P.Insertar(P1);
  P2->Captura();
  P.Insertar(P2);
  P3->Captura();
  P.Insertar(P3);
  E1->Captura();
  P.Insertar(E1);
  cout << "\n\n";
  while(!P.Vacia()) {
          Aux=P.Sacar();
          Aux->Muestra();
  }
  return 0;
}
```

En este ejemplo se implementa una clase Pila mediante un arreglo, en donde el tipo de la pila es un "parámetro" adicional, esto se logra por medio del preámbulo **template <class TIPO>**, en donde TIPO es el "parámetro" que indica el tipo de la pila.

La sintaxis general para definir una clase parametrizada es la siguiente:

```
template <class TIPO>
class Nombre_Clase {
        . . . . .
};
```

En el ejemplo anterior se definió una pila parametrizada mediante el siguiente código:

```
template <class TIPO>
class Pila {
        TIPO* Datos;
        int Top;
public:
        Pila() {
                Datos = new TIPO[MAX];
```

```
                Top = VACIO;
        }
        ~Pila() {delete [] Datos; }
        void Insertar(TIPO c);
        TIPO Sacar();
        int Vacia();
        int Llena();
};
```

Nótese que la definición de la clase está basada en el tipo TIPO el cual no se conoce aún, TIPO es solamente el nombre genérico del posible tipo de la clase Pila. Nótese también que el constructor y el destructor son métodos "inline" (están implementados en la definición de la clase) por lo que la sintaxis de la implementación es la usual. Sin embargo la implementación de los métodos que son "outline" (implementados fuera de la definición de la clase) tienen una sintaxis ligeramente diferente, que se muestra a continuación:

```
template <class TIPO>
Tipo_Retorno Nombre_Clase<TIPO>::Nombre_Metodo(. . .) {
        . . . . .
}
```

Por ejemplo, observe en el siguiente fragmento de código la implementación del método (con tipo parametrizado) Insertar de la clase Pila.

```
template <class TIPO>
void Pila<TIPO>::Insertar(TIPO c)  {
        Datos[++Top] = c;
}
```

Una vez implementada esta clase Pila se pueden declarar instancias de clase indincado el tipo particular de la instancia, por ejemplo se pueden declarar instancias que son Pilas de enteros, Pilas de Personas o incluso Pilas de punteros a Personas (para generar una Pila paramétrica polimórfica), como se muestra en forma respectiva en las siguientes declaraciones de variables:

```
Pila<int> x;            // Declara una Pila de enteros
Pila<Persona> P;        // Declara una Pila de Personas
Pila<Persona*> PP;      // Declara una Pila de punteros a                    // Personas
```

2.7.1 Compatibilidad entre tipos parametrizados

La compatibilidad entre tipos parametrizados no es tan clara como en los tipos usuales de C++, debido a que el compilador de C++ no puede hacer una conversión automática de tipos. Suponga que en el ejemplo 2.8 cambiamos el programa principal por el siguiente:

```
main() {
        Pila<int> x_int1;
         x_int1.Insertar(1);
         x_int1.Insertar(2);

         Pila<int> x_int2;
         x_int2.Insertar(3);
         x_int2.Insertar(-2);

         Pila<float> x_float;
         x_float.Insertar(1.3);
         x_float.Insertar(5.1);

         x_int1=x_int2;
         x_int1=x_float;          // Esta línea genera error
}
```

En el anterior fragmento de código se declaran tres pilas: dos pilas de tipo entero y la tercera de tipo flotante. La última línea de este programa genera error debido a que C++ no puede convertir en forma automática la pila de flotantes en una pila de enteros. Es posible asignar una pila de enteros a otra pila de enteros, como se hace con la instrucción:

```
x_int1=x_int2;
```

2.7.2 Los argumentos en clases parametrizadas

Los argumentos de un tipo (clase) parametrizado no están restringidos a clases, aunque esto es lo más común. Los parámetros de un tipo parametrizado pueden ser enteros, flotantes, nombres de funciones o incluso expresiones constantes. Por ejemplo en el siguiente fragmento de código se define una clase parametrizada Vector, en la que son parámetro el tipo de la pila y el tamaño del arreglo que almacenará el vector.

```
template <class TIPO, int n>
class Vector {
        TIPO Datos[n];
            .........
}
```

Dos instancias de clase son compatibles si sus argumentos en el parámetro TIPO son los mismos y si sus argumentos en n tienen el mismo valor, por ejemplo las siguientes instancias son compatibles:

```
Vector<float,100> V1;
Vector<float, 2*50> V2;
```

En el siguiente ejemplo mejoramos la implementación de la pila del ejemplo 2.8, para esto incorporamos un nuevo parámetro para el tamaño máximo de la pila.

🖫 Ejemplo 2.9: Implementación de una pila parametrizada con dos par

```
// E29.CPP
#include <iostream.h>
#include "pers26.hpp"
#define VACIO  -1

template <class TIPO,int Max>
class Pila {
        TIPO* Datos;
        int Top;
public:
```

```
        Pila() {
            Datos = new TIPO[Max];
            Top = VACIO;
        }
    ~Pila() {delete [] Datos; }
    void Insertar(TIPO c);
    TIPO Sacar();
    int Vacia();
    int Llena();
};

template <class TIPO,int Max>
void Pila<TIPO,Max>::Insertar(TIPO c)  {
  Datos[++Top] = c;
}

template <class TIPO,int Max>
TIPO Pila<TIPO,Max>::Sacar() {
    return (Datos[Top--]);
}

template <class TIPO,int Max>
int Pila<TIPO,Max>::Vacia() {
    return (Top==VACIO);
}

template <class TIPO,int Max>
int Pila<TIPO,Max>::Llena() {
    return (Top==Max-1);
}

main() {
  Pila<int,50> x;
  x.Insertar(1);
  x.Insertar(2);
  x.Insertar(3);
  while(!x.Vacia())
        cout << x.Sacar() << "\n";
  cout << "Termino" << "\n\n";

  Pila<Persona*,50> P;
```

```
      Persona *P1=new Persona();
  Estudiante *E1 = new Estudiante();
  Persona *Aux = new Persona();
  P1->Captura();
  P.Insertar(P1);
  E1->Captura();
  P.Insertar(E1);
  cout << "\n\n";
  while(!P.Vacia()) {
         Aux=P.Sacar();
         Aux->Muestra();
  }
  return 0;
}
```

La declaración de la clase debe incluir el nuevo parámetro, como se muestra en el siguiente fragmento:

```
template <class TIPO,int Max>
class Pila {
        TIPO* Datos;
        int Top;
public:
        Pila() {
                Datos = new TIPO[Max];
                Top = VACIO;
        }
        ~Pila() {delete [] Datos; }
        void Insertar(TIPO c);
        TIPO Sacar();
        int Vacia();
        int Llena();
};
```

Adicionalmente la sintaxis de la implementación de los métodos de la clase se hace bastante más tediosa, ya que se debe incluir un parámetro adicional en el encabezado de cada función (aunque esta función no lo utilice). Por ejemplo analice la función `Insertar(TIPO c)`:

```
template <class TIPO,int Max>
void Pila<TIPO,Max>::Insertar(TIPO c)  {
        Datos[++Top] = c;
}
```

2.7.3 Funciones con tipo parametrizado

Además de clases parametrizadas, C++ permite definir funciones con tipo parametrizado de manera muy similar. Una función parametrizada (template function) define en realidad una familia de funciones; cada una con un tipo particular, los parámetros con que la función es invocada determinan cual versión de la función será ejecutada.

En el siguiente fragmento de código se presenta una función parametrizada que calcula el mínimo entre dos elementos cuyo tipo es desconocido cuando la función es implementada.

```
// fun_para.cpp
#include <stdio.h>

template<class TIPO> TIPO Min(TIPO a, TIPO b) {
// Note que esta función retorna un elemento tipo TIPO
        if(a<b)
                return a;
        else
                return b;
}

int main() {
        char Ch1='O', Ch2='F', Res;
        int N1=100, N2=200, R;
        Res=Min(Ch1,Ch2);
        printf("El caracter menor es: %c\n",Res);
        R=Min(N1,N2);
        printf("El entero menor es: %d",R);
        return 0;
}
```

Para finalizar esta sección, surgen una serie de preguntas importantes Cuál es la diferencia entre tipos parametrizados y polimorfismo? Puede uno sustituir al otro? Cuál concepto es mejor o más potente?

Aunque estos dos conceptos son teóricamente diferentes, en algunas situaciones son prácticamente intercambiables.

Una función es polimórfica si uno o más de sus parámetros puede asumir en el momento de la invocación diferentes tipos de datos, los cuales en C++ deben derivar de una misma clase base. El parámetro formal de la función (el parámetro presente en la declaración de la función) debe ser declarado del tipo de la clase base. Así, cualquier función que tenga al menos un parámetro que sea un puntero a una clase, es una función polimórfica y podrá ser usada con diferentes tipos de datos.

Una función parametrizada es diseñada, definida e implementada con la generalidad en mente, por lo tanto escribir una función parametrizada implica un mayor grado de abstracción y de cuidado, para mantener la generalidad evitando la dependencia de un tipo de datos.

Las principales diferencias en C++ entre una función polimórfica y una función parametrizada son las siguientes:

1. Las funciones parametrizadas pueden trabajar con tipos aritméticos.
2. Las funciones polimórficas deben usar punteros.
3. La generalidad del polimorfismo se restringe a una jerarquía de clases.
4. La generalidad de las funciones parametrizadas es ilimitada.
5. Las funciones parametrizadas tienden a generar un archivo ejecutable más grande.

En la siguiente sección presentaremos una jerarquía de clases parametrizada denominada *Recipiente de Clases* o *Container Class* (como originalmente es denominada en inglés), en la cual se implementan las

estructuras de datos clásicas. Aprovecharemos esta jerarquía para implementar de nuevo la lista de Personas, Estudiantes, Personas y Asistentes.

2.8 El Recipiente de Clases (Container Classes)

Una de las principales virtudes de la programación orientada a objetos es la reutilización de código. Es muy común encontrar aplicaciones en las que se necesiten listas, arreglos, árboles y otra serie de estructuras de datos. Pero debemos implementar de nuevo estas estructuras de datos en cada nuevo proyecto de programación? La respuesta es no, la idea central de programación orientada a objetos es reutilizar bibliotecas de clases que ya tienen implementadas estas estructuras de datos.

C++ tiene dos recipientes de clases implementados por separado. La primera de estas es una jerarquía de clases cuya clase superior es la clase **Object**. Esta jerarquía de clases aparece desde la versión 2.0 de Borland C++ y tiene una estructura similar a la jerarquía de clases que provee el lenguaje SmallTalk, vea la Figura 2.12. La segunda es en realidad un conjunto de recipientes de clases parametrizadas que fue introducido en la versión 3.0 de Borland C++.

La jerarquía de clases basada en la clase Object tiende a desaparecer (debido a que el recipiente de clases parametrizado es mucho más potente), por lo que no se recomienda su uso en nuevos proyectos de programación. Debido a lo anterior en esta sección presentaremos únicamente el recipiente de clases parametrizadas.

Los recipientes de clases parametrizadas son conocidos como *BIDS*, de *Borland International Data Structures*. Los BIDS están divididos en dos categorías: Los *FDSs* (*Fundamental Data Structures*) y *ADTs* (*Abstract Data Structures*). Los recipientes FDSs implementan estructuras de datos básicas, como son: vectores, listas enlazadas, y listas doblemente enlazadas. Mientras que los recipientes ADTs implementan estructuras de datos de más alto nivel, como son: pilas, colas, "bags", diccionarios.

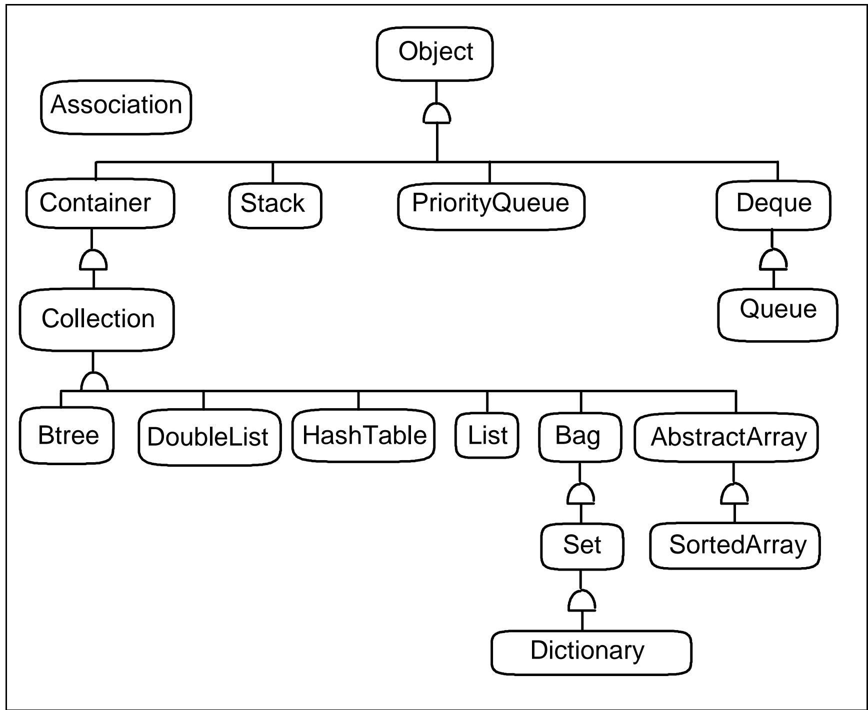


FIGURA 2.12. Recipiente de clases basado en la clase Object

En esta sección centraremos la atención en uno de los recipientes FDS, específicamente el recipiente FDS de listas (simplemente enlazadas - FDS List Container). Basta conocer con detalle la forma en que se reutiliza uno de estos recipientes de clases para entender cómo se pueden

reutilizar todos los recipientes de clases del BIDs. En la Figura 2.13 se presenta la jerarquía de clases FDS para listas simplemente (en una sola dirección) enlazadas .

Para analizar estas jerarquías es de mucha utilidad "*Browsing Objects*" de C++, el cual se puede activar con la opción View|Classes en el menú principal de Borland C++. Con esta herramienta el compilador de C++ presenta un diagrama de clases en el que se pueden consultar todas

las relaciones de herencia entre las clases, así como los métodos y atributos de cada clase, vea la Figura 2.14.

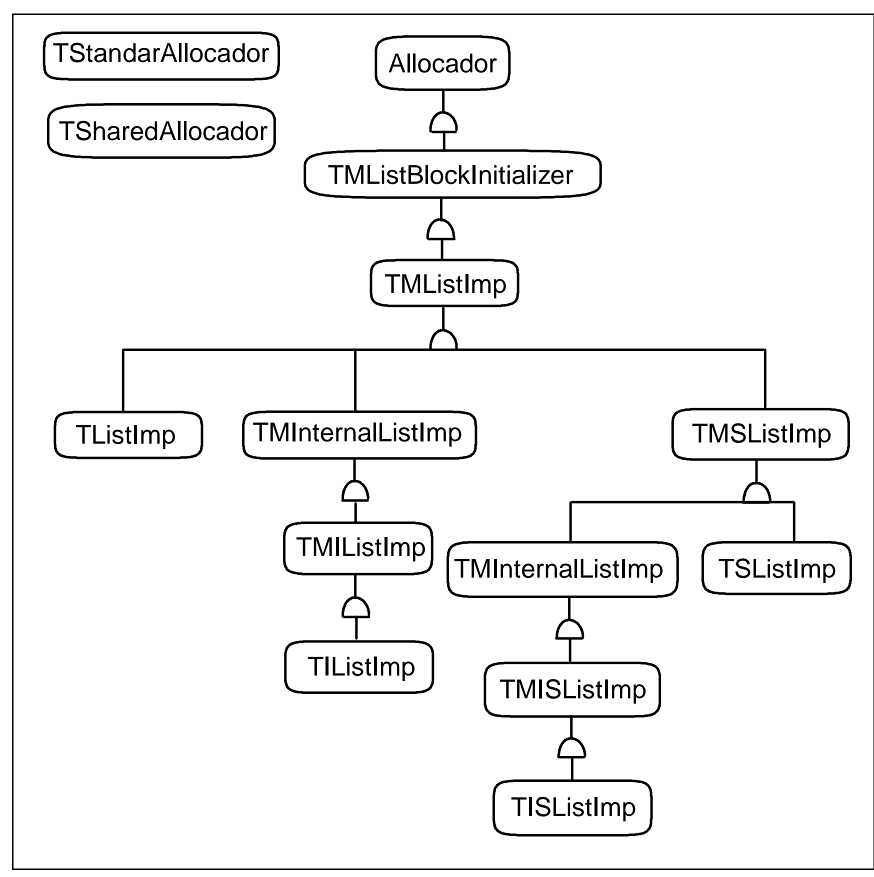


FIGURA 2.13. Jerarquía FDS para lista enlazadas

En el ejemplo 2.10 se ilustra cómo se puede reutilizar la clase TListImp. Para reutilizar una clase es importante conocer los métodos y atributos que la clase tiene, tanto propios como heredados. Con la

El Recipiente de Clases

ayuda de Borland C++ es fácil conocer los métodos y atributos de una clase, basta colocar el cursor sobre el nombre de la clase y presionar la tecla F1, con esto Borland C++ le presentará una ventana de ayuda en la que se describe la clase.

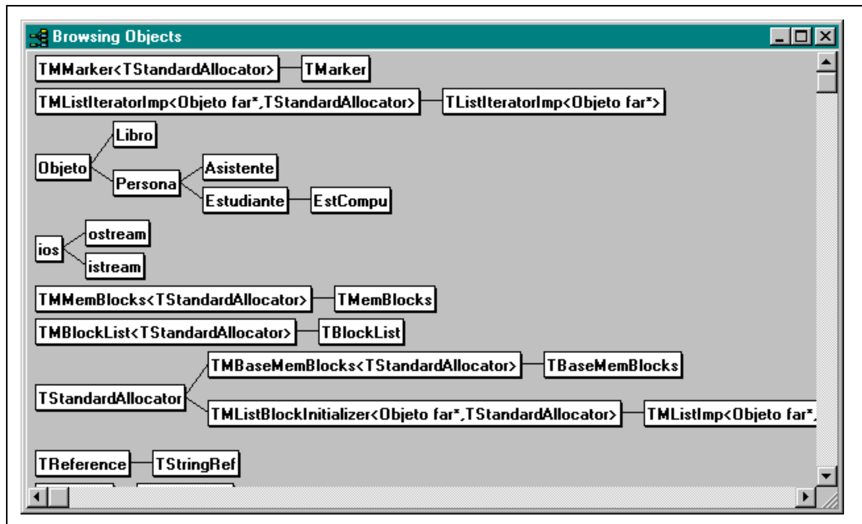


FIGURA 2.14 Browsing Object de Borland C++

Los métodos y atributos (algunos son heredados de la clase TMListImp) de la clase TListImp (ver el archivo de definición de la clase listimp.h) son los siguientes:

Constructor público

TMListImp()

Funciones miembro públicas

int Add(const T& t);
Agrega un objeto a la lista

int Detach(const T&, int = 0);

Remueve un objeto en el frente de la lista. Retorna 0 si falla y 1 si tiene éxito al remover el objeto. El segundo argumento especifica si el objeto debe ser borrado (ver TShouldDelete).

T *FirstThat(int (*)(const T &, void *), void *) const;
Retorna un puntero al primer objeto en la lista que satisface una condición dada. Usted suple una función-puntero f que retorna true para una condición verdadera. Se pueden pasar argumentos arbitrarios vía args. Retorna 0 si ningún objeto del arreglo conoce la condición.

void Flush(int del = 0);
Hace la lista vacía sin destruir esta.

void ForEach(IterFunc, void *);
Ejecuta una función f para la lista de elementos. ForEach crea un iterador interno para ejecutar la función dada para cada elemento en el arreglo.

int IsEmpty() const
Retorna 1 si la lista no tiene elementos; en otro caso retorna 0.

T *LastThat(int (*)(const T &, void *), void *) const;
Retorna un puntero al último objeto en la lista que satisface una condición dada. Usted suple una función-puntero f que retorna true para una condición verdadera. Se pueden pasar argumentos arbitrarios vía args. Retorna 0 si ningún objeto el arreglo conoce la condición.

Const T& PeekHead() const
Retorna una referencia a la cabeza de la lista, sin remover esta.

Funciones miembro protegidas

virtual TMListElement<T,Alloc> *FindDetach(const T& t)
virtual TMListElement<T,Alloc> *FindPred(const T&);

Atributos miembro protegidos

El Recipiente de Clases

TMListElement<T,Alloc> Head, Tail;

En el siguiente ejemplo se ilustra cómo se reutilizan algunas de estas funciones.

🖫 Ejemplo 2.10: Reutilización de la clase TListImp

```
// E2_10.CPP
#include <stdio.h>
#include <classlib\listimp.h>
#include "pers26.hpp"

main() {
  int x;
  TListImp<int> L;
  L.Add(1);
  L.Add(2);
  L.Add(3);
  while(!L.IsEmpty())  {
            x = L.PeekHead();        // Retorna la cabeza de la lista
            L.Detach(x);             // Remueve la primerta ocurrencia
            printf("El valor es: %d:\n",x);
  }
  printf("Termino\n\n");
  TListImp<Persona*> LP;
  Persona *P1=new Persona();
  Persona *P2=new Persona();
  Persona *P3=new Persona();
  Estudiante *E1 = new Estudiante();
  Persona *Aux = new Persona();
  P1->Captura();
  LP.Add(P1);
  P2->Captura();
  LP.Add(P2);
  P3->Captura();
  LP.Add(P3);
  E1->Captura();
  LP.Add(E1);
  printf("\n\n");
```

```
    while(!LP.IsEmpty())  {
        Aux = LP.PeekHead();  // Retorna la cabeza de la lista
        LP.Detach(Aux);       // Remueve la primera ocurrencia
        Aux->Muestra();
  }
  printf("Termino\n\n");
  return 0;
}
```

Obsérvese que en el ejemplo anterior se utiliza la clase TListImp para crear una lista de enteros y una lista de punteros a Personas, esto con las declaraciones:

```
TListImp<int> L;
TListImp<Persona*> LP;
```

El lector habrá notado que la función Add de la clase TListImp agrega siempre los elementos de la lista al inicio. Si desea que los elementos sean agregados al final de la lista entonces debe utilizar la clase TDoubleListImp que tiene métodos para agregar tanto al inicio de la lista como al final, estos métodos son int AddAtHead(const T& t) y int AddAtTai(const T&) respectivamente.

A continuación se implementa nuevamente el ejemplo 2.7, pero esta vez mediante la reutilización de la clase TLisImp. Como puede verse en este ejemplo, la cantidad de código programado por nosotros es mucho menor que la del ejemplo 2.7.

Nótese que la relación Com-Com entre la clase Estudiante y la clase Libro se implementa también utilizando la clase TLisImp. Analice con detalle las líneas de código resaltadas en negrilla e itálica.

Cada lista de la jerarquía FDS tiene asociada una clase que tiene la función de iterador o de un índice. En el siguiente fragmento de código se ilustra cómo se declara una lista con su iterador asociado.

```
TListImp<Persona*> UnaLista;
```

El Recipiente de Clases

```
TListIteratorImp<Persona*> Sig(UnaLista); // ITERADOR
```

La clase iterador asociada a la clase TListImp es la clase TListIteratorImp. Los métodos de TListIteratorImp son:

Un constructor público

```
TMListIteratorImp(const TMListImp<T,Alloc> &l)
```

Funciones miembro públicas

const T& Current()
Retorna el objeto actual.

void Restart()
Restaura el iterador al inicio de la lista.

Operadores

operator int();
Convierte el iterador a un valor entero para verificar si existen objetos en el resto de la lista. El iterador se convierte en 0 si no hay nada en el resto.

const T& operator ++ (int)
Mueve el iterador al siguiente objeto, y retorna el objeto en que estaba el iterador justo antes de moverlo (post-incremento).

const T& operator ++ ()
Mueve el iterador al siguiente objeto, y retorna el objeto en el que queda el iterador después de moverlo (pre-incremento).

La lista TDoubleListImp posee operadores operator--(int) y operador--(), los cuales permiten recorrer la lista hacia atrás.

🖫 Ejemplo 2.11: Lista de Personas mediante la reutilización de la clase TListImp

```
    // Programa de Prueba
// E2_11.CPP

#include "pers2_11.hpp"
#include <classlib\listimp.h>
#include <conio.h>
#include <stdio.h>

void LeeDatosPersona(void);
void LeeDatosAsistente(void);
void LeeDatosEstudiante(void);
void LeeDatosEstCompu(void);
char Menu(void);
Asistente *PAsistente;
Persona *PPersona;
Estudiante *PEstudiante;
EstCompu *PEstCompu;
FILE *Arch;

void main() {
// Las siguientes dos líneas sustituyen la declarción:
// ListaObjetos UnaLista;
  TListImp<Persona*> UnaLista;
  TListIteratorImp<Persona*> Sig(UnaLista);
  char opcion;
  do {
        clrscr();
        opcion = Menu();
        switch(opcion) {
              case '1': clrscr();
                        LeeDatosPersona();
                        UnaLista.Add(PPersona);
                        break;
              case '2': clrscr();
                        LeeDatosAsistente();
                        UnaLista.Add(PAsistente);
                        break;
              case '3': clrscr();
                        LeeDatosEstudiante();
                        UnaLista.Add(PEstudiante);
                        break;
```

El Recipiente de Clases

```
                        case '4': clrscr();
                                  LeeDatosEstCompu();
                                  UnaLista.Add(PEstCompu);
                                  break;
                        case '5': clrscr();
                         // El siguiente código sustituye a -> UnaLista.Muestra();
                                  Sig.Restart(); // Restaura el iterador
                                  while(Sig) {
                                            PPersona=Sig.Current();
                                             // Retorna un puntero a nodo actual
                                            PPersona->Muestra();
                                            Sig++;                                              // Incremer
                                  }
                                  getch();
                                  break;
            }
  } while(opcion!='6');
}

void LeeDatosPersona() {
  PPersona=new Persona();
  PPersona->Captura();
}

void LeeDatosEstudiante() {
  PEstudiante=new Estudiante();
  PEstudiante->Captura();
}

void LeeDatosEstCompu() {
  PEstCompu=new EstCompu();
  PEstCompu->Captura();
}

void LeeDatosAsistente() {
  PAsistente = new Asistente();
  PAsistente->Captura();
}

char Menu(void) {
```

```
        char opcion;
      printf("[1] Insertar Persona\n");
      printf("[2] Insertar Asistente\n");
      printf("[3] Insertar Estudiante\n");
      printf("[4] Insertar Estudiante de Computación\n");
      printf("[5] Ver la Lista Polimórfica\n");
      printf("[6] Salir\n");
      opcion= getch();
      return opcion;
}
```

// PERS2_11.HPP

```
// protege para múltiple inclusión
#if ! defined(_I_PERS2_11_HPP_)
#define _I_PERS2_11_HPP_

#include <stdio.h>
#include "objeto.hpp"
#include <classlib\listimp.h>
#define MAX1 30
#define MAX2 50

class Libro : public Objeto {
      char Nombre[30];
      int Anno;
public:
      Libro();
      virtual void Muestra(void);
      virtual void Captura(void);
      virtual void CambiaNombre(char NuevoNom[30]);
      virtual void CambiaAnno(int NuevoAnno);
      virtual void ObtNombre(char Nom[30]);
      virtual int  ObtAnno(void);
      virtual void Guardar(FILE *archivo);
      virtual void Recuperar(FILE *archivo);
};

class Persona : public Objeto {
protected:                // Permite que estos atributos sean
      char Nombre[30];    // conocidos en las clases derivadas
      int  Edad;
```

El Recipiente de Clases

```
public:
        Persona(char Nom[30], int Ed);
        Persona();
        virtual void Muestra(void);
        virtual void Captura(void);
        virtual void CambiaNombre(char NuevoNom[30]);
        virtual void CambiaEdad(int NuevaEdad);
        virtual void ObtNombre(char Nom[30]);
        virtual int  ObtEdad(void);
        virtual void Guardar(FILE *archivo);
        virtual void Recuperar(FILE *archivo);
};

class Estudiante : public Persona {
protected:
        float Examen1;
        float Examen2;
        float Promedio;
        int NumeroLibros;
        TListImp<Libro*> L;  // Relación Com-Com
public:
        Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
        Estudiante();
        virtual void  CalculaPromedio(void);
        virtual void  Muestra(void);
        virtual void  Captura(void);
        void  CambiaNotas(float NuevaNota1, float NuevaNota2);
        virtual float ObtEx1(void);
        virtual float ObtEx2(void);
        virtual float ObtProm(void);
        virtual void Guardar(FILE *archivo);
        virtual void Recuperar(FILE *archivo);
};

class EstCompu : public Estudiante {
        float  Examen3;
public:
        EstCompu(char Nom[30],int Ed, float Ex1, float Ex2, float Ex3);
        EstCompu();
        // No requiere método Muestra pues lo hereda sin ningún cambio
```

```
            virtual void Captura(void);
        void CambiaNotas(float NuevaNota1, float NuevaNota2, float Nueva-
Nota3);
        virtual void CalculaPromedio(void);
        virtual float ObtEx3(void);
        virtual void Guardar(FILE *archivo);
        virtual void Recuperar(FILE *archivo);
};


class Asistente : public Persona {
  float Promedio;
  int HorasAs;
  char Curso[50];
  char Carrera[50];
public:
  Asistente(char Nom[30],int Ed,float Prom,char Cur[50],int HA,char Carr[50]);
  Asistente();
  virtual void Muestra(void);
  virtual void Captura(void);
  virtual float ObtPromedio(void);
  virtual int  ObtHorasAs(void);
  virtual void ObtCurso(char Cur[50]);
  virtual void ObtCarrera(char Carr[50]);
  virtual void CambiaPromedio(float NuevoProm);
  virtual void CambiaHorasAs(int NuevaHo);
  virtual void CambiaCurso(char NuevoCurso[50]);
  virtual void CambiaCarrera(char NuevaCarrera[50]);
  virtual void Guardar(FILE *archivo);
  virtual void Recuperar(FILE *archivo);
};
#endif
```

// PERS2_11.CPP

```
#include "pers2_11.hpp"
#include <string.h>
#include <stdio.h>

/*------------------------P E R S O N A------------------------------*/

Persona::Persona(char Nom[30],int Ed) {
```

El Recipiente de Clases

```
  strcpy(Nombre,Nom);
  Edad = Ed;
}

Persona::Persona() {
  strcpy(Nombre,"");
  Edad = 0;
}

void Persona::Muestra(void) {
  printf("Persona: %s\n",Nombre);
  printf("Edad:    %d\n",Edad);
}

void  Persona::Captura(void){
  printf("Nombre:    ");
  flushall();
  gets(Nombre);
  printf("Edad: ");
  scanf("%d",&Edad);
}

void Persona::CambiaNombre(char NuevoNom[30]) {
  strcpy(Nombre,NuevoNom);
}

void Persona::CambiaEdad(int NuevaEdad) {
  Edad = NuevaEdad;
}

void Persona::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}

int Persona::ObtEdad(void) {
  return Edad;
}

void Persona::Guardar(FILE *archivo) {
  char aux='P';
```

```
        fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
}

void Persona::Recuperar(FILE *archivo) {
  fread(&Edad,sizeof(int),1,archivo);
  fread(Nombre,MAX1*sizeof(char),1,archivo);
}

/*------------------------L I B R O------------------------------*/

Libro::Libro() {
  strcpy(Nombre,"");
  Anno = 0;
}

void Libro::Muestra(void) {
  printf("Nombre del libro: %s\n",Nombre);
  printf("Año del libro:    %d\n",Anno);
}

void Libro::Captura(void) {
  printf("Nombre del Libro: ");
  flushall();
  gets(Nombre);
  printf("Año del Libro:    ");
  scanf("%d",&Anno);
}

void Libro::CambiaNombre(char NuevoNom[30]) {
  strcpy(Nombre,NuevoNom);
}

void Libro::CambiaAnno(int NuevoAnno) {
  Anno = NuevoAnno;
}

void Libro::ObtNombre(char Nom[30]) {
  strcpy(Nom,Nombre);
}
```

El Recipiente de Clases

```
int Libro::ObtAnno(void) {
  return Anno;
}

void Libro::Guardar(FILE *archivo) {
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Anno,sizeof(int),1,archivo);
}

void Libro::Recuperar(FILE *archivo) {
  fread(Nombre,MAX1*sizeof(char),1,archivo);
  fread(&Anno,sizeof(int),1,archivo);
}

/*------------------------E S T U D I A N T E---------------------------*/

// Llama al constructor de la clase base
Estudiante::Estudiante(char Nom[30],int Ed,float Ex1,float Ex2) : Perso-
na(Nom,Ed) {
  Examen1 = Ex1;
  Examen2 = Ex2;
}

Estudiante::Estudiante() : Persona() {
  Examen1 = 0;
  Examen2 = 0;
}

void Estudiante::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2)/2;
}

void Estudiante::Muestra(void)  {
  printf("Estudiante: %s\n",Nombre);
  printf("Promedio  : %f\n",Promedio);
  Libro *PLibro;
  TListIteratorImp<Libro*> Sig(L);
  while(Sig) {
            PLibro=Sig.Current();
```

```
                        PLibro->Muestra();
                  Sig++;
  }
}

// Programación Incremental
void Estudiante::Captura(void)  {
  Persona::Captura();                // Invoca a Captura de la clase base
  printf("Nota Examen 1: ");
  scanf("%f",&Examen1);
  printf("Nota Examen 2: ");
  scanf("%f",&Examen2);
  Promedio=(Examen1+Examen2)/2;
  printf("Número de libros del estudiante: ");
  scanf("%d",&NumeroLibros);
  Libro *PLibro;
  for(int i=1; i<=NumeroLibros; i++) {
                  PLibro=new Libro();
                  PLibro->Captura();
                  L.Add(PLibro);
  }
}

void Estudiante::CambiaNotas(float NuevaNota1, float NuevaNota2) {
  Examen1 = NuevaNota1;
  Examen2 = NuevaNota2;
}

float Estudiante::ObtEx1(void) {
  return Examen1;
}

float Estudiante::ObtEx2(void) {
  return Examen1;
}

float Estudiante::ObtProm(void) {
  return Promedio;
}

void Estudiante::Guardar(FILE *archivo) {
```

El Recipiente de Clases

```
  char aux='E';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Examen1,sizeof(float),1,archivo);
  fwrite(&Examen2,sizeof(float),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
}

void Estudiante::Recuperar(FILE *archivo) {
  Persona::Recuperar(archivo);
  fread(&Examen1,sizeof(float),1,archivo);
  fread(&Examen2,sizeof(float),1,archivo);
  fread(&Promedio,sizeof(float),1,archivo);
}

/*------------------------E S T - C O M P---------------------------*/

// Llama al constructor de la clase base
EstCompu::EstCompu(char Nom[30],int Ed,float Ex1,float Ex2,float Ex3) : Es-
tudiante(Nom,Ed,Ex1,Ex2) {
          Examen3 = Ex3;
}

void EstCompu::Captura(void)  {
  Persona::Captura();              // Invoca a Captura de la clase Persona
  printf("Nota Examen 1: ");       // No invoca a Captura de la clase base
  scanf("%f",&Examen1);            // pues debe capturar 3 exámenes y no 2
  printf("Nota Examen 2: ");       // y porque la forma de calcular el promedio
  scanf("%f",&Examen2);            // es diferente
  printf("Nota Examen 3: ");
  scanf("%f",&Examen3);
  Promedio=(Examen1+Examen2+Examen3)/3;
  printf("Número de libros del estudiante: ");
  scanf("%d",&NumeroLibros);
  Libro *PLibro;
  for(int i=1; i<=NumeroLibros; i++) {
                    PLibro=new Libro();
                    PLibro->Captura();
                    L.Add(PLibro);
```

```
        }
}

EstCompu::EstCompu() : Estudiante() {
            Examen3 = 0;
}

void EstCompu::CambiaNotas(float NuevaNota1, float NuevaNota2, float Nue-
vaNota3) {
  Examen1 = NuevaNota1;
  Examen2 = NuevaNota2;
  Examen3 = NuevaNota3;
}


void EstCompu::CalculaPromedio(void) {
  Promedio = (Examen1 + Examen2 + Examen3)/3;
}

float EstCompu::ObtEx3(void) {
  return Examen3;
}

void EstCompu::Guardar(FILE *archivo) {
  char aux='C';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Examen1,sizeof(float),1,archivo);
  fwrite(&Examen2,sizeof(float),1,archivo);
  fwrite(&Examen3,sizeof(float),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
}

void EstCompu::Recuperar(FILE *archivo) {
  Persona::Recuperar(archivo);
  fread(&Examen1,sizeof(float),1,archivo);
  fread(&Examen2,sizeof(float),1,archivo);
  fread(&Examen3,sizeof(float),1,archivo);
  fread(&Promedio,sizeof(float),1,archivo);
}
```

El Recipiente de Clases

```
/*------------------------A S I S T E N T E-----------------------------*/

Asistente::Asistente(char Nom[30],int Ed,float Prom,char Cur[50],
                                    int HA,char Carr[50]) : Persona(Nom,Ed){
   Promedio = Prom;
            strcpy(Curso,Cur);
   HorasAs = HA;
   strcpy(Carrera,Carr);
}

Asistente::Asistente(){
            Promedio = 0;
   strcpy(Curso,"");
   HorasAs = 0;
   strcpy(Carrera,"");
}

void Asistente::Muestra(void) {
  printf("Asistente:         %s\n",Nombre);
  printf("Edad:              %d\n",Edad);
  printf("Carrera:           %s\n",Carrera);
  printf("Horas Asignadas:   %d\n",HorasAs);
  printf("Curso Asignado:    %s\n",Curso);
  printf("Promedio Ponderado %f\n",Promedio);
}

void Asistente::Captura(void) {
  Persona::Captura();
  printf("Carrera:           ");
  flushall();
  gets(Carrera);
  printf("Horas Asignadas:   ");
  scanf("%d",&HorasAs);
  printf("Curso Asignado:    ");
  flushall();
  gets(Curso);
  printf("Promedio Ponderado ");
  scanf("%f",&Promedio);
}
```

```
float Asistente::ObtPromedio(void) {
  return Promedio;
}

int Asistente::ObtHorasAs(void) {
  return HorasAs;
}

void Asistente::ObtCurso(char Cur[50]) {
  strcpy(Cur,Curso);
}

void Asistente::ObtCarrera(char Carr[50]) {
  strcpy(Carr,Carrera);
}

void Asistente::CambiaPromedio(float NuevoProm) {
  Promedio = NuevoProm;
}

void Asistente::CambiaHorasAs(int NuevaHo) {
  HorasAs = NuevaHo;
}

void Asistente::CambiaCarrera(char NuevaCarrera[50]) {
  strcpy(Carrera,NuevaCarrera);
}

void Asistente::CambiaCurso(char NuevoCurso[50]) {
  strcpy(Curso,NuevoCurso);
}

void Asistente::Guardar(FILE *archivo) {
  char aux='A';
  fwrite(&aux,sizeof(char),1,archivo);
  fwrite(&Edad,sizeof(int),1,archivo);
  fwrite(Nombre,MAX1*sizeof(char),1,archivo);
  fwrite(&Promedio,sizeof(float),1,archivo);
  fwrite(&HorasAs,sizeof(int),1,archivo);
  fwrite(Curso,MAX2*sizeof(char),1,archivo);
```

El Recipiente de Clases

```
  fwrite(Carrera,MAX2*sizeof(char),1,archivo);
}

void Asistente::Recuperar(FILE *archivo) {
  Persona::Recuperar(archivo);
  fread(&Promedio,sizeof(float),1,archivo);
  fread(&HorasAs,sizeof(int),1,archivo);
  fread(Curso,MAX2*sizeof(char),1,archivo);
  fread(Carrera,MAX2*sizeof(char),1,archivo);
}
```

En el siguiente fragmento de código del ejemplo anterior se ilustra cómo es utilizado el iterador para mostrar el contenido de la lista:

```
// El siguiente código sustituye a -> UnaLista.Muestra();
        Sig.Restart(); // Restaura el iterador
        while(Sig) {
                PPersona=Sig.Current();
                 // Retorna un puntero a nodo actual
                PPersona->Muestra();
                Sig++;                                   // Incremer
        }
        getch();
        break;
```

Como ya habíamos mencionado la relación com-com entre la clase Estudiante y la clase Libro se implementa mediante la clase TListImp, esto puede verse claramente en el siguiente fragmento de código de la definición de la clase Estudiante:

```
class Estudiante : public Persona {
protected:
        float Examen1;
        float Examen2;
        float Promedio;
        int NumeroLibros;
        TListImp<Libro*> L;  // Relación Com-Com
public:
```

```
        Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
    Estudiante();
    . . . . . . . . .
    . . . . . . . . .
};
```

Cuando se muestra a un Estudiante se debe también mostrar su lista de libros, para esto se declara un iterador asociado a la variable L de la clase Estudiante y luego mediante un ciclo se recorre la lista para desplegarla. Lo anterior se presenta en el siguiente fragmento de código:

```
void Estudiante::Muestra(void)  {
  printf("Estudiante: %s\n",Nombre);
  printf("Promedio  : %f\n",Promedio);
  Libro *PLibro;
  TListIteratorImp<Libro*> Sig(L);
  while(Sig) {
            PLibro=Sig.Current();
            PLibro->Muestra();
            Sig++;
  }
}
```

El único problema de reutilizar la clase TlistImp para implementar nuestra lista de Personas es que esta lista no tiene métodos para guardar y recuperar sus elementos del disco. Para corregir este problema, en el ejemplo 2.12 se implementa la clase ListaObjetos la cual reutiliza la clase TListImp y agrega los métodos Guardar y Recuperar. Lo anterior se hizo mediante una relación Com-Com, pero también puede hacerse mediante una relación de herencia.

Seguidamente presentamos el ejemplo 2.12, en este se incluye únicamente el código de la nueva implementación de la clase ListaObjetos, pues tanto el código de la jerarquía de Personas, Estudiantes, EstuCompu y Asistentes como el código de programa principal son idénticos a los que se presentan el ejemplo 2.7.

🖫 Ejemplo 2.12: Implementación de la clase ListaObjetos me- diante

El Recipiente de Clases

```
// LPER2_12.HPP
// protege para múltiple inclusión
#if ! defined(_I_LPER2_12_HPP_)
#define _I_LPER2_12_HPP_

#include "objeto.hpp"
#include <classlib\listimp.h>

class ListaObjetos {
	TListImp<Objeto*> L;
public:
	void Inserta(Objeto *NuevaPersona);
	void Muestra(void);
	void Guardar(FILE *archivo);
	void Recuperar(FILE *archivo);
	void Guardar(FILE *archivo,int N);
	void Recuperar(FILE *archivo,int &N);
};
#endif

// LPER2_12.CPP
#include <stdio.h>
#include "lper2_12.hpp"
#include "pers2_12.hpp"

void ListaObjetos::Inserta(Objeto *NuevaPersona) {
	L.Add(NuevaPersona);
}

void ListaObjetos::Muestra() {
	TListIteratorImp<Objeto*> Sig(L);
	Objeto *PObjeto;
	while(Sig) {
		PObjeto=Sig.Current(); // Retorna un puntero a nodo actual
		PObjeto->Muestra();
		Sig++;			// Incrementa el iterador
	}
}

void ListaObjetos::Guardar(FILE *archivo)  {
```

```
        archivo = fopen("personas.dat","w");
    Objeto *PObjeto;
    TListIteratorImp<Objeto*> Sig(L); // Define el iterador
    Sig.Restart();
    while(Sig) {
        PObjeto=Sig.Current(); // Retorna un puntero a nodo actual
        PObjeto->Guardar(archivo);
        Sig++;                  // Incrementa el iterador
    }
  fclose(archivo);
}

void ListaObjetos::Recuperar(FILE *archivo)  {
  char clave;
  L.Flush();
  archivo = fopen("personas.dat","r");
  while(!feof(archivo)) {
    if(fread(&clave,sizeof(char),1,archivo)!=0) {
        if(clave=='P') {
                Persona *PObjeto=new Persona();
                PObjeto->Recuperar(archivo);
                Inserta(PObjeto);
        }
        if(clave=='E') {
                Estudiante *PEstudiante=new Estudiante();
                PEstudiante->Recuperar(archivo);
                Inserta(PEstudiante);
        }
        if(clave=='C') {
                EstCompu *PEstCompu=new EstCompu();
                PEstCompu->Recuperar(archivo);
                Inserta(PEstCompu);
        }
        if(clave=='A') {
                Asistente *PAsistente=new Asistente();
                PAsistente->Recuperar(archivo);
                Inserta(PAsistente);
        }
        if(clave=='L') {
            Libro *PLibro=new Libro();
            PLibro->Recuperar(archivo);
```

El Recipiente de Clases

```
                        Inserta(PLibro);
                }
        }
 }
 fclose(archivo);
}

void ListaObjetos::Recuperar(FILE *archivo,int &N)  {
 if(fread(&N,sizeof(int),1,archivo)!=0) {
        for(int i=1; i<=N; i++) {
                        Libro *PLibro=new Libro();
                        PLibro->Recuperar(archivo);
                        Inserta(PLibro);
        }
 }
}

void ListaObjetos::Guardar(FILE *archivo,int N)  {
 TListIteratorImp<Objeto*> Sig(L);
 Objeto *PObjeto;
 if(fwrite(&N,sizeof(int),1,archivo)!=0) {
        while(Sig) {
                PObjeto=Sig.Current(); // Retorna un puntero a nodo actual
                PObjeto->Guardar(archivo);
                Sig++;                      // Incrementa el iterador
        }
 }
}
```

2.9 Constructores y destructores

El objetivo central

de los *constructores* y *destructores* en la Programación Orientada a Objetos es que los "tipos definidos por el programador", es decir las clases, se comporten de la misma forma que los tipos de datos propios del lenguaje. Esto quiere decir que cuando en un programa se declara una instancia de clase (una variable cuyo tipo es de alguna clase definida

por el programador), su memoria sea reservada y sus atributos sean inicializados en forma automática, y que cuando el ámbito de validez ("scope") de la instancia sea superado la memoria sea liberada automáticamente, tal como sucede con los tipos de datos propios de lenguaje.

Para entender cómo y cuándo los constructores son invocados se deben conocer y entender muy bien las reglas para la definición e implementación de este tipo especial de funciones, además se debe tener claro también cómo se comportan estas funciones en relaciones de herencia y de Componente-Compuesto.

Obviamente, como cualquier otra función en C++, los constructores y los destructores deben ser definidos e implementados antes de que puedan ser usados, pero adicionalmente la definición e implementación tienen ciertas reglas que son únicas en este tipo de funciones.

Las reglas más importantes para definir un ***constructor*** son las siguientes:

Un constructor debe tener el mismo nombre que el de la clase a la que pertenece, por ejemplo la clase class Persona { ...} debe tener como constructor la función Persona().

Un constructor no debe tener tipo de retorno. El compilador Borland C++ genera un tipo especial de retorno invisible para el programador; por ejemplo:

```
class Persona {
        . . .
public:
  P
ersona(char Nom[30], int Ed);
        // Constructor, no tiene tipo de retorno
        . . .
};
```

El código de un constructor puede hacerse dentro de la misma definición de la clase, esto se conoce como código "inline", pero solamente se recomienda cuando el código es muy corto y simple, por ejemplo:

```
class Persona {
        . . .
public:
  Persona(char Nom[30], int Ed) {
        strcpy(Nombre,Nom);  // constructor con código inline.
        Edad = Ed;
  }
        . . .
};
```

El código de un constructor puede hacerse también fuera de la definición de la clase, esto se conoce como código "out-of-line", esta es la forma usual de implementar el constructor, por ejemplo:

```
class Persona {
        . . .
public:
  Persona(char Nom[30], int Ed); // Constructor out-of-line
        . . .
};
        . . .

Persona::Persona(char Nom[30],int Ed) {
 strcpy(Nombre,Nom);
 Edad = Ed;
}
```

Como un constructor puede tener argumentos, la sobrecarga es permitida, es decir una clase puede tener uno o más constructores, por ejemplo:

```
class Persona {
        . . .
public:
```

```
      Persona(char Nom[30], int Ed);
   Persona();
          . . .
};
```

Un constructor no debe ser heredado. Si un constructor es heredado puede causar una serie de problemas inesperados, ya que cuando se declara un objeto de una clase derivada, entonces el constructor de la clase base es invocado primero y luego el constructor de la clase derivada; si el constructor es heredado, entonces la clase base no tendrá constructor lo cual causará una gran confusión al compilador.

Las reglas más importantes para definir un ***destructor*** son muy similares a las de los constructores; son las siguientes:

Un destructor tiene el mismo nombre de la clase, excepto que es precedido por el símbolo ~, por ejemplo:

```
class Persona {
          . . .
public:
  Persona(char Nom[30], int Ed);        // Constructor
 ~Persona();                            // Destructor
          . . .
};
```

Un destructor puede tener código "inline" o "out-of-line". Si el código es "out-of-line" entonces la sintaxis es como se muestra en el siguiente ejemplo:

```
Persona::~Persona()  {
          . . . // Código del destructor
}
```

Un destructor no debe tener tipo de retorno; al igual que el constructor, el compilador genera automáticamente un tipo de retorno propio del destructor.

A diferencia del constructor, un destructor no debe tener argumentos, esto implica que los destructores no pueden ser sobrecargados por lo que una clase tendrá necesariamente un único destructor.

Un destructor no puede ser heredado, esto por cuanto causaría los mismos problemas ya explicados para el caso del constructor, es decir que el destructor correcto no necesariamente será el invocado.

El destructor puede ser una función virtual, esto ayuda algunas veces a que sea invocada una versión correcta del destructor, sin embargo no es muy común el uso de destructores virtuales.

Ahora bien, surgen dos preguntas importantes:

1. Cuándo son invocados los constructores? Los constructores son invocados automáticamente cuando un objeto de la clase es creado, así los constructores no son ejecutados por el compilador, si cunado se declara una variable de tipo clase, es decir cuando una instancia de clase es creada en tiempo de ejecución, por ejemplo:

```
class Persona {
        . . .
};
        . . .
void main() {
        Persona A;              // Invoca a constructor sin parámetros
        Persona B("Luis",30);   // Invoca a constructor con parámetros
        . . .
};
```

Otra forma usual de invocar un constructor es a través del operador ***new***, cuando en lugar de declarar una instancia de clase se declara una variable tipo puntero a la clase, por ejemplo:

```
class Persona {
        . . .
};
        . . .
void main() {
```

```
      Persona* PA;                  // Declara un puntero a la clase.
   PA = new Persona;        // Crea el objeto e invoca al constructor.
   . . .
   Persona *PB;                  // Declara un puntero a la clase.
   PB = Persona("Luis",30); // Crea el objeto e invoca al constructor                              // con par
   . . .
};
```

La declaración del puntero y la invocación del constructor se puede hacer en una sola instrucción como se muestra en el siguiente ejemplo:

```
class Persona {
      . . .
};
      . . .
void main() {
      Persona* PA =  new Persona;
            // Declara un puntero a la clase, crea el objeto e
            // invoca al constructor sin parámetros.
      . . .
      Persona *PB = Persona("Luis",30);
            // Declara un puntero a la clase, crea el objeto e
            //  invoca al constructor con parámetros.
      . . .
};
```

Lo anterior logra que las clases C++ se comporten de la misma forma que los tipos propios de C, pues para declarar, por ejemplo una variable tipo entero, o bien un puntero a un entero, se puede hacer como sigue:

```
      void main() {
            int n;
            int *pn;  // Puntero a un entero.
            pn = new int;
            . . .
      }
```

Como se puede ver, es exactamente la misma sintaxis que se usó para declarar variables tipo Persona. Obviamente las variables tipo clase, al igual que los tipos propios del lenguaje, pueden ser creados en el cuerpo de cualquier otra función definida por programador o inclusive en funciones que pertenecen a otras clases.

2. Cuándo son invocados los destructores? Los destructores funcionan de forma casi inversa a los constructores. El destructor es invocado automáticamente cuando el ámbito ("scope") de la instancia (variable tipo clase) termina, en otras palabras cuando el bloque donde es declarada la variable termina, por ejemplo:

```
if(x>0) {
    Persona A;
        . . .
}       // En esta llave termina el ámbito de A, por lo que el       // destructo
```

Cuando una instancia u objeto es declarado en el cuerpo de una función, el destructor es invocado automáticamente después de que el mecanismo de retorno (return) es ejecutado, por ejemplo:

```
void FuncionEjemplo(. . . )  {
        Persona A;
        . . .
};      // Aquí es invocado el destructor de Persona.
```

Otra situación común se da cuando se declara un arreglo de objetos como el siguiente:

```
Persona  LP[100];
```

entonces el constructor es invocado en orden ascendente de acuerdo con la dirección de memoria y los destructores son invocados en orden inverso.

Los objetos creados con operador ***new*** tienen duración arbitraria por lo que para invocar el destructor se debe usar el operador ***delete***, como se muestra en el siguiente ejemplo:

```
class Persona {
        . . .
};
        . . .
void main() {
        Persona* PA =  new Persona;
        . . .
        delete PA;
        . . .
};
```

Ciertamente también se pueden crear arreglos arbitrarios de objetos con el operador new, los cuales deben ser destruidos a través del operador delete, como se muestra en el siguiente ejemplo:

```
class Persona {
        . . .
};
        . . .
void main() {
        Persona* PA =  new Persona[22];
        . . .
        delete PA [22];
        . . .
};
```

Finalmente, el destructor también puede ser invocado en forma explícita para un objeto dado. En el siguiente ejemplo se ilustra la sintaxis usada para invocar al destructor explícitamente:

```
class Persona {
        . . .
};
        . . .
```

```
void main() {
    Persona PA("Luis",30);
    . . .
    PA.Persona::~Persona();        // Invocación al destructor.
    . . .
};
```

ahora bien, si lo que se ha declarado es una variable tipo puntero a una clase entonces la sintaxis para invocar el destructor en forma explícita es la siguiente:

```
class Persona {
        . . .
};
        . . .
void main() {
        Persona* PA =  new Persona;
        . . .
        PA->Persona::~Persona();      // invocación del destructor a través
        . . .                         // de un puntero.
};
```

Las tareas principales de un constructor son inicializar los atributos de la clase y reservar memoria para la clase. Mientras que la tarea principal del destructor es liberar la memoria que está utilizando la clase; el mejor ejemplo es el constructor y destructor de la lista enlazada presentada en el ejemplo 2.5 cuyo código es el siguiente:

```
struct Nodo {
   Persona *PPersona;
   Nodo    *Sig;
};

class ListaPersonas {
  Nodo *Primera;
public:
  ListaPersonas();
  ~ListaPersonas();
  void Inserta(Persona *NuevaPersona);
```

```
    void Muestra(void);
};
        . . .
ListaPersonas::ListaPersonas() {
  Primera = NULL;
}

ListaPersonas::~ListaPersonas() {
  // Este ciclo libera la memoria de la lista
  while(Primera != NULL) {
    Nodo *Tempo=Primera;
    Primera=Primera->Sig;
    delete Tempo->PPersona;
    delete Tempo;
  }
}
```

Constructores y destructores en relaciones Com-Com

Cuando una clase contiene un miembro que a su vez es una instancia de clase, entonces el constructor del objeto miembro (componente) es ejecutado "antes" que el constructor del objeto compuesto, es decir lo que sucede realmente es que antes de que se ejecute la primera línea del código del constructor del objeto compuesto, se ejecuta el constructor del objeto componente. Veamos el siguiente fragmento de código tomado del ejemplo 2.5.

```
class Libro {
   . . .
public:
  Libro();
   . . .
};
        . . . . .
class Estudiante : public Persona {
   . . .
  Libro L;              // Relación Com-Com
public:
  Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
```

```
      Estudiante();
  . . .
};

void  main()  {
        Estudiante  E; // Se invoca al constructor
        . . .
}
```

Como puede verse, la clase Estudiante tiene un componente de tipo Libro, por lo que cuando la variable E es declarada en la función main() el constructor de la clase Estudiante es invocado automáticamente, pero antes de que se ejecute la primera instrucción de este constructor, el constructor de la clase Libro es invocado en forma automática. La mejor forma de entender lo que sucede es "correr" el programa usando en Debugger paso a paso.

Los destructores actúan de forma inversa, es decir, el cuerpo del destructor del objeto compuesto es ejecutado antes de que el destructor del objeto componente sea ejecutado. Por ejemplo en el siguiente fragmento de código el destructor es invocado cuando la sentencia delete E es ejecutada; aquí el destructor de la clase Estudiante se ejecuta antes que el destructor de la clase Libro.

```
class Libro {
  . . .
public:
  Libro();
  ~Libro();
  . . .
};
        . . . . .
class Estudiante : public Persona {
  . . .
  Libro L;                    // Relación Com-Com
public:
  Estudiante(char Nom[30], int Ed, float Ex1, float Ex2);
  Estudiante();
  ~Estudiante();
  . . .
```

```
};

void   main()  {
        Estudiante  *E = new Estudiante; // Se invoca al constructor
        . . .
        delete E; . . . . . . . .            // Se invoca al destructor
}
```

Constructores y destructores en relaciones de herencia

El orden en que los constructores se invocan es claro, debido a que C++ exige que el constructor de la clase derivada invoque al constructor de la clase base, por lo que el constructor de la clase base se ejecuta primero que el de la clase derivada, con esto queremos decir, que el constructor de la clase base se ejecuta antes de que el cuerpo del constructor de la clase derivada se ejecute. Así, por ejemplo, el constructor de la clase Estudiante invoca explícitamente a constructor de la clase Persona, por lo que este último constructor se invoca primero. Esto puede verse en el siguiente fragmento de código:

```
Estudiante::Estudiante() : Persona() {
        Examen1 = 0;
        Examen2 = 0;
}
```

El orden en que los destructores son invocados en relaciones de herencia es muy simple, se invocan en un orden inverso al orden en que los constructores son invocados, o sea, que el destructor de la clase derivada se invoca antes que el destructor de la clase base.