Unidad 3. Programación Orientada a Objetos

Es una metodología de diseño de software y un paradigma de programación en el que define a los programas en términos de "clases de objetos", objetos que son entidades que combinan su estado (es decir, datos) y su comportamiento (esto es, procedimientos o funciones). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, y que a su vez se comunican entre ellos para realizar tareas. Esto difiere de los lenguajes procedurales tradicionales, en los que los datos y los procedimientos están separados y sin relación. Los programas diseñados con orientación a objetos están pensados para ser códigos más simples de mantener y reutilizar.

La programación orientada a objetos anima al programador a pensar los programas principalmente en términos de tipos de datos, y en segundo lugar en las operaciones (métodos) específicas que interactúan con esos datos. Mientras que los lenguajes procedurales animan al programador a pensar sobre todo en términos de procedimientos, y en segundo lugar en los datos que ésos procedimientos manejan.

1 Conceptos de Programación Orientada a Objetos

El paradigma orientado a objetos se enfoca a las características de comportamiento y estructura de las entidades como unidades completas, lo que nos permite diseñar software de manera modular y con un alto manejo de estructuras de datos complejas. El paradigma de programación orientado a objetos se apoya en los siguientes conceptos:

• **Abstracción** (de datos) involucra la formulación de un concepto (clase) poniendo atención en las similitudes y diferencias entre las entidades de un conjunto, para extraer las características esenciales que lo distingan y evitar las características no relevantes. Y así, se establece una sola representación del concepto que tenga esas características pertinentes.

Es la capacidad de crear tipos de datos definidos por el usuario extrayendo las propiedades esenciales de un concepto, sin preocuparse de los detalles exactos de la implementación. Algunos simplemente lo definen como la capacidad de enfocarse en lo esencial.

• **Encapsulación** asegura que los usuarios de un objeto no alteren de manera inesperada el contenido de un objeto; solo sus métodos internos están autorizados para acceder a ellos y su interfaz pública especifica como otros objetos podrían interactuar con él.

La abstracción y la encapsulación están representadas por la clase. La clase es una abstracción, porque define los atributos y métodos de un determinado conjunto de objetos con características comunes, y es una encapsulación por que constituye una caja negra que encierra tanto los datos que almacena como los métodos que permiten manipularlos.

• **Polimorfismo** es la propiedad que permite a una operación tener diferente comportamiento en diferentes objetos. Es decir, diferentes objetos reaccionan de manera diferente al mismo mensaje (de modo que un mismo método pueda tener múltiples implementaciones).

• **Herencia** es la capacidad de definir una nueva clase a partir de otra. De esta forma, la reutilización de código esta garantizada. Las clases están clasificadas en una jerarquía estricta, donde la clase padre es conocida como superclase y la clase hijo como clase derivada. Esto significa que la clase derivada dispone de todos los atributos y métodos de su superclase.

Los anteriores conceptos son los pilares fundamentales de la programación orientada a objetos y describen de manera precisa su funcionamiento. La construcción de programas orientados a objetos permite facilitar la comunicación, incrementar la productividad y la consistencia en los modelos computacionales, así como facilitar la modificación de los programas y reducir la complejidad y el esfuerzo de resolución de problemas.

1.1 Definición de clase, atributo, método y objeto

Recuerde que las clases son abstracciones de entidades presentes en un dominio determinado de problemas, y que las clases están compuestas de diversos componentes que involucran el comportamiento del modelo a plasmar, así pues definiremos brevemente esos componentes.

Una clase es un tipo de dato definido por el usuario (dato abstracto), es muy similar a una estructura, ya que especifica los datos que se verán involucrados en la construcción del programa, pero además deberá contemplar la implementación de las funciones que deberán interactuar con dichos datos. Con frecuencia se dice que la clase es equivalente a la generalización de un objeto.

Se dice que los datos o atributos contenidos en la clase son el conjunto de características que definen las propiedades de un objeto especifico, generalmente se conocen como variables del objeto. Y por otro lado, las funciones o métodos son pequeños programas altamente independientes asociados a los datos específicos de la clase. De manera que éstos son los componentes necesarios para construir un objeto cualquiera.

Así pues, los objetos son cajas negras que mantienen ocultas propiedades estructurales llamadas atributos, y propiedades de comportamiento llamadas métodos. De esta forma, los objetos son entonces instancias o ejemplares de una clase.

Como una forma de estandarizar la nomenclatura de clases, se especifican dos reglas muy sencillas. Los nombres de las clases deberán iniciar con mayúscula y escribirse en singular, y los nombres de los atributos y los métodos deberán escribirse preferentemente con minúsculas.

2 Abstracción de Datos

La Abstracción de datos involucra la formulación de un concepto (clase) poniendo atención en las similitudes y diferencias entre las entidades de un conjunto, para extraer las características esenciales que lo distingan y evitar las características no relevantes. Y así, se establece una sola representación del concepto que tenga esas características pertinentes. Algunos simplemente lo definen como la capacidad de enfocarse en lo esencial.

2.1 Tipos de datos definidos por el usuario en C (datos abstractos)

Es muy común emplear las estructuras de datos como tipos de datos definidos por el usuario, además de que resultan muy útiles ya que a partir de ellas podemos definir un nuevo tipo de dato que podrá estar compuesto de elementos de otros tipos y en algunos casos quizás de más estructuras. Por otro lado, las funciones juegan un papel trascendental en la definición de datos, ya que es posible construir funciones que manejen esos tipos de datos definidos por el usuario, lo que puede aumentar y simplificar el diseño de programas ya que las funciones pueden estar asociadas a un tipo de dato en especial.

En el siguiente ejemplo se muestra como podemos construir un programa a partir de estructuras, y después desarrollar las funciones necesarias para poder interactuar con dichas estructuras. Construimos una estructura llamada Horaria que define 3 valores hora, minuto, segundo. El programa principal genera una variable de tipo estructura Horario a la cual se le asignan valores iniciales y que posteriormente es enviada por referencia (&) a las funciones Militar() y Standar(), que se encargan de mostrar la hora en formato universal y estándar.

```
#include <iostream>

using namespace std;

/* tipo Horario definido por el usuario mediante una estructura simple */
struct Horario {
        int hora, minuto, segundo;
};

/* declaración de prototipos */
void Militar ( Horario & ); // reciben una referencia por “alias”
void Standar ( Horario & );

int main(){
        Horario Mexico; // variable tipo Horario
        Mexico.hora = 14;       // asignan valores a los miembros
        Mexico.minuto = 57;
        Mexico.segundo = 0;
```

```
        cout << "La hora en Mexico es ";
        Militar( Mexico ); /* envia como parámetro la variable Mexico */
        cout << " formato militar, \n es decir las ";
        Standar( Mexico );
        Mexico.hora = 29;       // asigna un valor inválido al miembro cout << "\n datos invalidos
        ";
        Militar( Mexico );
        cout << endl;
        return 0;
}

/* definición de funciones prototipo */

/* recibe como argumento la referencia a la variable tipo Horario */
void Militar ( Horario &t ){
// imprime los miembros de la estructura
        t. cout << ( t.hora < 10 ? "0":"" ) << t.hora << ( t.minuto < 10 ? "0":"" ) << t.minuto;
/* Recuerde que el operador condicional es ( condición ? valor-cierto : valor-falso ) */
}

/* recibe como argumento la referencia de la variable tipo Horario */
void Standar ( Horario &t ){
        cout << ( ( t.hora == 0 || t.hora == 12 ) ? 12 : t.hora % 12 )
        << ":" << ( t.minuto < 10 ? "0":"" ) << t.minuto
        << ":" << (t.segundo < 10 ? "0":"" ) << t.segundo
        << ( t.hora < 12 ? " AM" : " PM");
}
```

2.2 Tipos de datos abstractos en C++

En la programación estructurada los programas se dividen en dos componentes: procedimientos y datos. Este método permite empaquetar código de programa en procedimientos, pero ¿Qué sucede con los datos?, las estructuras de datos utilizadas en programación estructurada son globales o se pasan como parámetros, por lo que en esencia los datos se manipulan de manera separada de los procedimientos.

En el paradigma Orientado a Objetos los programas se consideran como un solo componente, en el que están contenidos tanto los procedimientos, como los datos. De manera que cada componente que se genere es catalogado como un objeto. Así pues, un objeto es una unidad que contiene datos y funciones que operan sobre esos datos. De manera tal que a las variables de un objeto se les conoce como miembros o propiedades, y a las funciones que operan sobre los datos se les denominan métodos.

En los lenguajes de programación orientados a objetos, la clase es la unidad básica de programación y los objetos son variables o instancias de la misma clase que inevitablemente deberán de entablar una comunicación entre sí mediante el paso o envío de mensajes (acciones

que debe ejecutar el objeto) a través de los métodos. El paso de mensajes es el término utilizado para referirnos a la invocación o llamada de una función miembro de un objeto.

Cada vez que se construye un objeto de una clase, se crea una instancia de esa clase. En general, los términos objetos e instancias de una clase se pueden utilizar indistintamente. Una clase es una colección de objetos similares (generalización) y un objeto es una instancia o ejemplar de una definición de una clase.

Las clases, en lenguaje C++, son el tipo de dato abstracto que nos permite realizar el proceso de abstracción de manera eficiente ya que nos abocamos al diseño especifico de la clase esbozando sus características mas importantes describiendo únicamente los prototipos de las funciones miembro (declaración) y posteriormente desarrollamos la implementación de cada una de las funciones miembro para esa clase en especial (definición).

Para el diseño de clases en C++ emplearemos la palabra clave class y especificamos el nombre de la clase, posteriormente declaramos los miembros y los métodos (funciones miembro) de la clase. Adicionalmente por el momento emplearemos dos etiquetas que nos indicaran el tipo de acceso a los miembros de la clase, public y private. Los datos miembro public (públicos) serán vistos desde cualquier punto fuera de la clase y los datos private (privados) estarán únicamente disponibles por los métodos pertenecientes a la clase. A continuación se presenta la estructura gramatical de la declaración e implementación de clases en C++:

```
// declaración del dato abstracto

class <Nombre-clase>
{
        // declaración de variables miembro o propiedades
        [private:|public:|protected:]
        tipo <lista-de-variables>;

// declaración de funciones miembro o métodos
        [private:|public:|protected:]
        <tipo-de-regreso> <función-miembro>([lista-argumentos]);
};

// implementación de la función

<tipo-de-regreso> <Nombre-clase>::<función-miembro>([lista-argumentos])
{
        // declaración de variables locales
        // cuerpo de la función
}
```

3 Construcción de Clases y objetos

Retomaremos el ejemplo visto en la sección anterior y lo replantearemos para su solución mediante clases. Primeramente diseñaremos la estructura indispensable para su funcionamiento

(declaración), empleando la técnica de abstracción, y nos ocuparemos de colocar los elementos indispensables para su funcionamiento.

3.1. Componentes de una Clase

Son todos aquellos datos y funciones (elementos) que forman parte de un determinado objeto, y que representan sus características y propiedades particulares. La clase encapsula estos elementos y los dota de ciertos criterios de visibilidad y manipulación con respecto a otras clases.

3.2. Datos y Funciones miembro

•Datos miembro: Son todas la variables asociadas a una clase particular y representan las propiedades o características más básicas de un objeto.

•Funciones miembro: Son todas las funciones o métodos que están relacionadas con un objeto particular, es decir, son todas aquellas acciones que puede realizar un objeto de cierta clase y que por lo general manipulan sus propiedades.

Inicialmente necesitamos declarar un nombre para la clase, en este caso emplearemos el mismo nombre de la estructura Horario. Ahora recordemos que la estructura Horario estaba compuesta de 3 elementos (hora, minuto, segundo), para este caso dejaremos restringido el acceso a estos datos a través de la etiqueta private, ya que no deseamos que estos datos sean manipulados por algún usuario o clase externa.

De forma similar declararemos las funciones (prototipos) que deseamos implementar para la clase, únicamente se declaran los prototipos de las funciones y su tipo de acceso, y más adelante se implementara el mecanismo o funcionamiento para cada función miembro.

El método constructor, como su nombre lo indica, es la función miembro responsable de construir la infraestructura (datos) necesaria para que un objeto este disponible. Es una función que lleva exactamente el mismo nombre de la clase y es la primera función en ejecutarse de manera automática cuando se crea una instancia de la clase.

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

/***     declaración      ***/
class Horario {
private: // identificador de acceso restringido a variables
        /*** datos miembro ***/
        int hora, minuto, segundo;

public: // identificador de acceso libre a métodos
        Horario(); // método constructor
        void ajuste(int, int, int);/*** métodos miembro (prototipos) ***/
        void Militar();
        void Standar();
```

```
};
```

3.3 Implementación de Funciones miembro

Consisten en desarrollar el mecanismo de funcionamiento para cada función miembro, es decir, es la parte de programación de la función. Aquí es donde realmente se realiza el proceso de programación, es importante aclarar que todos los datos y funciones miembro de una clase son alcanzados de manera global a la clase. De esta manera cualquier función miembro de una clase podrá acceder a cualquiera de sus miembros, incluyendo a los elementos privados.

A continuación implementamos las funciones miembro para la clase que acabamos de declarar. Note que el método constructor no tiene tipo, además tome en cuenta que en la implementación de los métodos es necesario colocar primero el nombre de la clase y después el operador " :: " que es el operador de pertenencia (alcance) en la implementación, seguido irá el nombre del método que será implementado, incluyendo los argumentos que recibirá.

```
/***     definición de la clase     ***/

Horario::Horario()          //Constructor: inicializa la estructura del objeto
{
        hora = minuto = segundo = 0;
}

/* método "ajuste" asigna una nuevo horario valido */
void Horario::ajuste( int h, int m, int s)
{
        hora = ( h >= 0 && h < 24 ) ? h : 0; minuto = ( m >= 0 && m < 60 ) ? m : 0; segundo = ( s >= 0
        && s < 60 ) ? s : 0;
}

/* método Militar, imprime la hora en formato militar de 24 hrs */
void Horario::Militar()   {
        cout << ( hora < 10 ? "0":"" ) << hora
        << ( minuto < 10 ? "0":"" ) << minuto;
}

/* método Estándar, imprime la hora en formato estándar AM/PM */
void Horario::Standar() {
        cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 )
        << ":" << ( minuto < 10 ? "0":"" ) << minuto
        << ":" << (segundo < 10 ? "0":"" ) << segundo
        << ( hora < 12 ? " AM" : " PM");
}
```

Observe que los métodos miembro realizan exactamente la mismas instrucciones que las funciones desarrolladas en el programa estructurado. A excepción de la función miembro ajuste

que fue incorporada en el mecanismo de la clase para que nos permitiera realizar la modificación de los datos miembro privados (hora, minuto y segundo).

Creación y Uso de Objetos

Ahora veremos como emplear la clase Horario en un programa, debe tomar en cuenta que lo que ahora haremos será definir variables del tipo Horario y de esta manera crear objetos (instancias) de la clase Horario. Una vez declarado un objeto, el método constructor será el primer método en ejecutarse inmediatamente, de tal manera que el constructor es el encargado de crear la infraestructura necesaria para que el objeto funcione. Una vez creado, los métodos públicos del objeto estarán disponibles, y pueden ser accedidos de manera muy similar a las estructuras empleando el operador miembro "." (punto), como veremos a continuación:

```
int main(){
        Horario Mexico; /* declaramos un objeto de la clase Horario, enseguida se ejecutara el
        método constructor Horario() */
        cout << "Hora inicial ";
        Mexico.Militar(); // llamamos a uno de los métodos
        cout << " es decir ";
        Mexico.Standar();
        cout << "\n Nuevo horario en México ";
        int h, m, s;
        cin >> h >> m >> s;
        Mexico.ajuste(h, m, s); // modificamos los datos usando ajuste()
        cout << "\nLa hora en Mexico es ";
        Mexico.Standar();       // llamamos a uno de los métodos del objeto
        system("pause");
        return 0;
}
```

Los métodos constructores inicializan variables y en la mayoría de los casos crean la infraestructura necesaria para que el objeto pueda trabajar con sus propios datos, por ejemplo inicialización de variables y vectores dinámicos. Una vez que el objeto a cumplido su labor y ya no es necesario que ocupe espacio de memoria, será ejecutado automáticamente método destructor, que se encargará de liberar el espacio de datos en memoria pertenecientes al objeto.

Por lo general los métodos destructores son mas usados cuando generamos espacio dinámico y requerimos del destructor para que realice la liberación de memoria de las variables dinámicas. Mas adelante veremos ejemplos mas claros y efectivos de la utilización del método destructor.

3.4 Constructor y Destructor
3.5 Tipos de Constructores

Como ya se explico, los constructores son el primer método que se ejecuta al crear un objeto de esa clase, tiene la labor de inicializar las variables y crear toda la estructura necesaria para que los objetos de una determinada clase funcionen como se pretende.

En los lenguajes orientados a objetos como C++ existe la posibilidad de que una misma función pueda ser declarada varias veces, con la única restricción de que el número de argumentos puede variar desde ningún argumento hasta n. A esta característica de las funciones se le llama sobrecarga de funciones, y es muy útil cuando se necesita inicializar un objeto de varias maneras. Veamos un ejemplo nuevamente con la clase Horario que hemos empleado:

```
class Horario {
private:
        int hora, minuto, segundo;
public:
        Horario(); // constructor por defecto
        Horario(int, int, int);      // constructor general
        void ajuste(int, int, int);
        void Militar();
        void Standar();  };

/* constructor por defecto, sin argumentos */
        Horario::Horario() {
        hora = minuto = segundo = 0;
}

/* constructor general, con argumentos */
        Horario::Horario(int h, int m, int s) {
        ajuste(h,m,s);
}
```

Una vez definidas las funciones constructoras, estamos en posibilidad de generar objetos de la clase Horario. Ahora, lo que haremos será declarar un par de objetos de éste tipo y emplearemos dos modos diferentes de declaración para cada objeto; uno para un objeto llamado mex el cual no cuenta con argumentos y otro llamado usa con argumentos de entrada. El objeto mex será construido empleando el constructor sin argumentos, mientras que el objeto usa será construido empleando el constructor con argumentos. El compilador es el encargado de manejar la sobrecarga, ya que en base al número de argumentos del objeto, el compilador determinará a que función le corresponde encargarse de la construcción del objeto.

```
int main( void )
{
        Horario mex, usa(13,30,0);
        cout << " Horario en México "; mex.Standar(); // 00:00:00 AM cout << " Horario en USA ";
        usa.Standar(); // 01:30:00 PM system("pause");
        return 0;
}
```

También es posible tener constructores que tengan valores predeterminados, y así podemos crear un solo constructor que permita recibir ninguno o varios argumentos. Cada argumento que ingrese será asignado de izquierda a derecha en los argumentos de la implementación del constructor de la clase, los valores que no sean ingresados serán asignados según el valor de la declaración.

```
class Horario {
private:
        int hora, minuto, segundo;

public:
        Horario(int = 0, int = 0, int = 0); // constructor predeterminado
        void ajuste(int, int, int);
        void Militar();
        void Standar();
};
        Horario::Horario(int h, int m, int s) { // constructor predeterminado
        ajuste(h,m,s);
}
```

De esta forma podremos declarar objetos con el numero de argumentos que se requiera para el constructor, por ejemplo podríamos decidir que unos objetos inicialicen sin argumentos y otros con uno, dos o tres argumentos para el caso de la clase Horario.

```
int main( void )
{
        Horario mex, usa(13), can(15,1,1);
        cout << " Horario en México "; mex.Standar();
        //00:00:00 AM
        cout << " Horario en USA "; usa.Standar();
        //01:00:00 PM cout << " Horario en Canada "; can.Standar();
        //03:01:01 PM
        system("pause");
        return 0;
}
```

Además de los constructores antes mencionados existe un tipo de constructor llamado constructor de copia, que consiste en crear un nuevo objeto a partir de otro objeto ya existente. Para ello sera necesario asignar todas las variables miembro del objeto existente al objeto de recién construcción.

```
class Horario {
private:
        int hora, minuto, segundo;
public:
        Horario(int = 0, int = 0, int = 0);
        Horario( const Horario & ); // constructor de copia
        void ajuste(int, int, int);
        void Militar();
```

```
        void Standar();
};
Horario::Horario( const Horario &x ) { // constructor de copia
hora = x.hora; // asigna los valores x al propietario de la llamada
minuto = x.minuto;
segundo = x.segundo;
}
```

Los constructores de copia son muy útiles cuando necesitamos construir una replica exacta de un objeto ya definido previamente, por lo que su construcción depende de un objeto como argumento.

```
int main( void )
{
Horario mex(13,30), usa( mex ); // usa se construye a partir de mex
cout << " Horario en México "; mex.Standar();// 01:30:00 PM cout << " Horario en USA "; usa.Standar();// 01:30:00 PM cin.get();
return 0;
}
```

Como ya hemos visto, las formas de diseñar los constructores varia según las necesidades y creatividad del programador. Una buena abstracción del problema y un eficiente diseño de las funciones miembro pueden resultar en el desarrollo de una clase eficiente, funcional y optima, que cubra las necesidades para las que fue desarrollada.

Ejemplo: A continuación veremos una clase llamada Vector, e implementaremos un par de constructores (predeterminado y de copia), además veremos la manera en como puede ser implementado el método destructor que como se menciono anteriormente, es la contraparte de los constructores, y tienen el propósito de liberar el espacio dinámico. Los destructores son ejecutados una vez que se ha perdido el alcance de un objeto, es decir son la ultima función que se ejecutará.

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

class Vector {
private:
        int nelem; // tamaño del vector
        int * lista; // puntero para el vector dinámico public:
        Vector( int = 1 ); // inicializa con 1 elemento
        Vector( const Vector & ); // constructor de copia
        ~Vector(); // destructor
        void capturar(); // captura los elementos del vector
        void mostrar();
        void ordenar();
        void suma( const Vector &, const Vector & ); // suma de vectores
};

/* constructor predeterminado */
```

```
Vector::Vector( int x ) {
        nelem = x; // asignamos a n el numero de elementos
        lista = new int [ nelem ]; // construimos el vector dinámico
        for( int i = 0; i < nelem; i++ )
        lista[ i ] = 0; // inicializamos el vector
}

/* crea una copia exacta de un objeto a partir de otro ya existente */

Vector::Vector( const Vector &x ) {
        nelem = x.nelem;
        lista = new int [ nelem ];
        for( int i = 0; i < nelem; i++ )
        lista[ i ] = x.lista[ i ];
}

Vector::~Vector() { // destructor
        delete [] lista; // libera espacio asignado de manera dinámica
}
```

Ahora implementamos el método captura, que es responsable de ingresar cada elemento dentro del vector dinámico, observe que la variable nelem sirve como delimitador de elementos.

```
void Vector::capturar() {           /*** captura los elementos del vector ***/
        for ( int i = 0; i < nelem; i++ ) {
                cout << “elemento “ << i;
                cin >> lista[ i ];
        }
}
```

Por otro lado, en ocasiones necesitamos que un objeto interactué con otros objetos, ya sea de su mismo tipo ú otro. Para ello es necesario diseñar algunos métodos que nos permitan realizar operaciones más interesantes. Por ejemplo piense que un objeto de tipo Vector, además de almacenar información del número de elementos, cuenta con la característica de poder realizar operaciones entre ellos; así pues veremos como sería la implementación de un procedimiento suma para vectores del mismo tipo y tamaño.

```
void Vector::suma(const Vector &A, const Vector &B ){
if ( A.nelem == B.nelem ) {// verifica que sean iguales
        delete [] this->lista; // eliminamos la lista local del objeto this->nelem = A.nelem;
        // asignamos el nuevo nelem de A o B lista = new int [nelem];
        // creamos un nuevo espacio
        for( int i = 0; i < nelem; i++ ) // pasamos los elementos de A y B
                lista[i] = A.lista[i] + B.lista[i];
                // los elementos de lista local se actualizan
} else
        lista[0] = -1; // no definida
}
```

El procedimiento suma recibe 2 objetos de tipo Vector como argumentos de entrada, de manera que dichos objetos pueden ser manipulados de forma transparente por el procedimiento suma, así que sus propiedades pueden ser vistas de manera local. Ahora es importante recordar que el objeto que ejecuta el procedimiento suma es considerado como el objeto propietario de la llamada por lo que sus propiedades no necesariamente deben ser referenciadas, pero es posible hacerlo, empleando el puntero this. De manera que el puntero this es la dirección de un objeto pasado como parámetro implícito a la función miembro.

Así que es posible referirnos a los miembros del objeto propietario de la llamada con el puntero this ó prescindiendo de él, mas adelante veremos a detalle la utilidad del puntero this. Prestemos ahora atención a la operación de suma realizada dentro del procedimiento y observe que cada elemento del vector A es sumado con cada elemento del vector B y el resultado es depositado en la lista de valores del propietario de la llamada.

De esta manera podríamos crear un objeto de tipo Vector para manipular sus datos y más adelante en caso de necesitar una copia exacta del objeto lo haríamos a través de su constructor de copia, inclusive es posible realizar una suma de ambos vectores y depositarlo en un tercero llamado C. Al finalizar el programa el método destructor se encargara automáticamente de liberar el espacio asignado a cada objeto de tipo Vector.

```
int main( void )
{
        Vector R(10), C; R.capturar(); R.mostrar();
        Vector S( R ); // crea una copia exacta de R en S
        C.suma( R , S ); // realiza una suma de R y S en C
        C.mostrar(); system(“pause”); return 0;
}
```

Nota: Se deja como ejercicio para el lector implementar los métodos mostrar (que imprime los elementos del vector) y ordenar (que ordena los elementos bajo el método de selección).