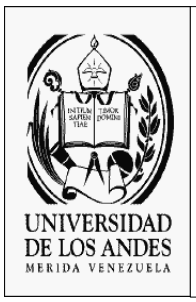


UNIVERSIDAD DE LOS ANDES
NUCLEO UNIVERSITARIO RAFAEL RANGEL (NURR)
DEPARTAMENTO DE FISICA Y MATEMATICA
AREA COMPUTACION
TRUJILLO EDO. TRUJILLO

Programacion II
POO
Programación Orientada a Objetos

PROFESOR: JOSE DELPHIN

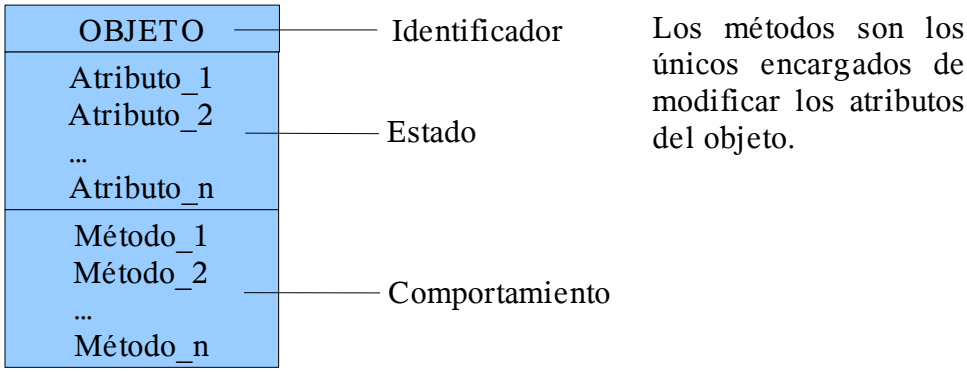
La programación orientada por objetos (POO)

Es un estilo de programación, basado en el uso de entidades u objetos, los cuales responden a ciertos estímulos que cambian el estado de los mismos, permitiendo así la interacción entre diversos objetos.

Qué es un Objeto?

Es una entidad que combina tres aspectos fundamentales, el estado, el comportamiento y la identidad.

- El *estado* se refiere a los datos, es decir, uno o varios atributos a los que se habrán asignado unos valores concretos.
- El *comportamiento* está definido por los procedimientos o métodos con que puede operar dicho objeto, es decir, qué operaciones se pueden realizar con él.
- La *identidad* es una propiedad de un objeto que lo diferencia del resto, dicho con otras palabras, es su identificador.



Característica de la POO.

Abstracción: Se refiere a las características esenciales de un objeto que permiten generalizar algo. Cuando se habla de una mujer, se piensa en una persona de sexo femenino, posiblemente con cabello largo, joven, etc. cuando se habla de un hombre, se piensa igualmente en una persona pero con características diferentes; en este caso se puede extraer un término que es común para ambos seres, y es el término Persona, sin tener en cuenta las características que diferencian a una de otra; ese es el término abstracto.

Encapsulamiento: esto es reunir todos los elementos suficientes y necesarios de una misma entidad, para logra una mayor cohesión de los componentes del sistema.

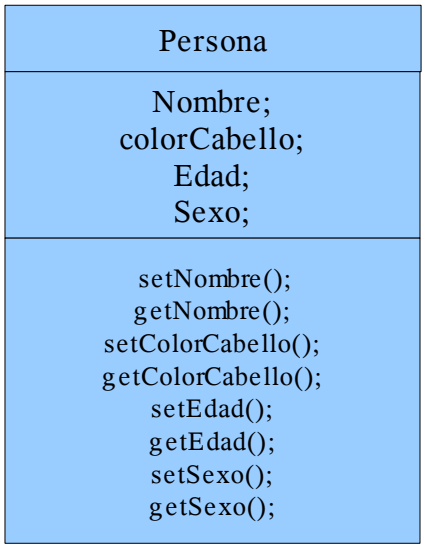
Principio de ocultación: Todos los atributos de un objeto son protegidos para que solo puedan ser accedidos por sus métodos. Con esto se asegura que otros objetos no puedan cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas.

Polimorfismo: se refiere a comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama *asignación tardía* o *asignación dinámica*. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las

plantillas y la sobrecarga de operadores de C++.

Herencia: los objetos no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia permite la reutilización de código, ya que los objetos de mayor jerarquía pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en *clases*.

En ***C++***, la abstracción y el encapsulamiento esta representado por una ***clase***, la clase es una abstracción por lo que se definen las propiedades (funciones) y atributos (datos) de un determinado conjunto de objetos.




```
class Persona{
  //atributos
  char* nombre;
  char* colorCabello[20];
  int edad;
  char sexo;

  public:
  //métodos
  int setNombre(char* elNombre, int tam);
  char* getNombre();
  ...
  ...
}
```

El cuerpo de los métodos de la clase, se realiza fuero de la definición de la misma, bajo la siguiente sintaxis:

<Tipo de retorno> <Nombre de clase>::<Nombre del método>(lista de parámetros){
  Cuerpo de la función o método.
}

Para el ejemplo tratado se vería de la siguiente forma:

```
int Persona::setNombre(char* elNombre, int tam){
    nombre = new char[tam];
    if (nombre == NULL){
        cout << "No hay espacio de memoria.";
        return -1;
    }
    *nombre = *elNombre;
    return 0;
}

char* Persona::getNombre(){
    return nombre;
}
```

Toda clase posee dos (2) tipos de métodos especiales: los constructores y los destructores. El constructor es llamado por el compilador cada vez que se crea un objeto de ese tipo, dicho de otra forma, el objeto es instanciado y el destructor es llamado siempre que una instancia de la clase se destruye.

Generalmente los constructores se utilizan para inicializar los atributos de la clase, y llevan el mismo nombre y el destructor para vaciar espacios de memoria, tambien lleva el nombre de la clase precedido del símbolo ~. En nuestro ejemplo se vería como sigue:

```
class Persona{
  //atributos
  char* nombre;
  char* colorCabello[20];
  int edad;
  char sexo;

  public:
  //métodos
  Persona();          //Constructor
  int setNombre(char* elNombre, int tam);
  char* getNombre();
  …
  …
  ~Persona();         //Destructor
}
```

Estructura Lineal de Datos

Una estructura lineal de datos o lista está conformada por ninguno, uno o varios elementos que tienen una relación de adyacencia ordenada donde existe un primer elemento, seguido de un segundo elemento y así sucesivamente hasta llegar al último. El tipo de dato de los elementos puede ser cualquiera, pero debe ser el mismo tipo para todos.

Clasificación de las Estructuras Lineales de Datos

Se pueden clasificarse según el criterio o base de la forma y el lugar que se sigue para realizar sus operaciones básicas (agregar, eliminar elementos, etc). De acuerdo a estos criterios se tiene:

De acuerdo al Criterio de las Operaciones de inserción y eliminación

- **Listas de Acceso Restringido**
  - ***Pilas:*** las operaciones de acceso se realizan por un único extremo de la lista (tope de la pila),
  - ***Colas:*** las operaciones se realizan por ambos extremos de la lista (se inserta por un extremo y se elimina por el otro) .
  - ***Dipolo:***
    - *Entrada Restringida*: Se inserta por un extremo y se elimina por ambos
    - *Salida Restringida:* Se elimina por un extremo y se inserta por ambos
- **Listas de Acceso no Restringido**
  - ***Listas generales:*** no tienen una restricción en cuanto a sus operaciones. se considera como la superclase de las otras clases de listas, en específico de las pilas, colas y dipolos.

De acuerdo al orden de sus elementos
- Ordenadas
  - Ascendentes
  - Descendentes
- Desordenadas

De acuerdo al método de almacenamiento
- Secuencial (Simple y Circular)
  - Elementos contiguos físicamente en memoria
  - Se utiliza el tipo arreglo
  - Son estáticas, porque su tamaño esta limitado por el tamaño del arreglo.
  - Acceso a los elementos en forma directa
- Enlazado (Simple Simple Circular Doble Doble Circular)
  - Manejo Dinámico de la memoria.
  - Se maneja mediante apuntadores
  - No tienen tamaño fijo

Representación Lógica

Las listas se representan en forma lógica como una secuencia de elementos.
Ejemplos: Una lista de Estudiantes, una lista de empleados, Una lista de artículos, etc.

En forma gráfica, una lista se representa como:

e_1	e_2	...	e_{n-2}	e_{n-1}	e_n

Estructuras de Almacenamiento para las estructuras lineales de datos

Para almacenar en memoria una lista se utilizan dos métodos: el secuencial y el enlazado

Método Secuencial: Los elementos de la lista están contiguos físicamente en la

memoria y para su soporte se utiliza el tipo Arreglo. Los accesos a los elementos se efectúan en forma directa, por medio de un índice, siendo el acceso mas rápido, sin embargo, el crecimiento de la mismas esta sujeto al tamaño máximo del arreglo.

Método de Enlazado: Los elementos en la lista no necesariamente se encuentran contiguos en la memoria. Las listas, pueden ampliar o limitar su tamaño mientras se ejecuta el programa.

Especificación del Tipo Abstracto de Datos LISTA

Las listas, representan el tipo más general, al cual se considera como la superclase de las otras clases de listas, en específico de las pilas, colas y dipolos. Haciendo la jerarquía de clases adecuada para estas estructuras, se tiene que la lista es la clase raíz de la jerarquía que tiene como subclases la pila, la cola y el dipolo.

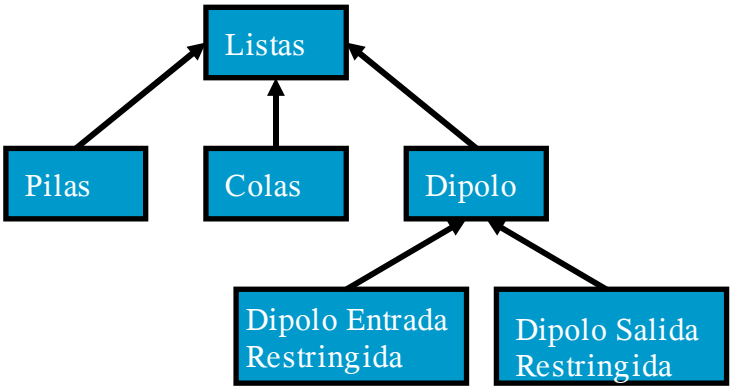


Figura No. 1: Jerarquía de Clases para las Listas

Propiedades de las Listas:

Cada elemento de la lista tiene asignado un tipo de dato. Si L representa un tipo de ListaDe[TipoEle: e] entonces e_1, e_2, ..., e_n conforman la lista cuyos elementos tienen asignado un mismo tipo.

Si n=0 entonces la lista esta vacía

Si n>=1 entonces e_1 es el primer elemento de la lista y e_n el ultimo

e_i es el predecesor de e_{i+1} y el sucesor de e_{i-1} con 1<=i<=n

Especificación del Tipo Abstracto de Datos LISTA

Especificación Lista[TipoEle]		
1 2 3	**Especificación Sintáctica** CreaLista()→Lista, InsLista()→Lista, EliLista()→Lista, ConsLista()→TipoEle, VaciaLista()→Logico, DestruyeLista()→. **Declaraciones** TipoEle : e, {TipoEleNoDef} **Especificación Semántica** VaciaLista(CrearLista())=Cierto VaciaLista(InsLista(CreaLista(),e))=Falso ConsLista(CrearLista())={TipoEleNoDef} EliLista(CrearLista())=CrearLista()	**CrearLista()** Crea una Lista vacía **InsLista()** Ingresa un nuevo elemento a la Lista según una posición especifica **EliLista(),** Elimina un elemento de la Lista según una posición especifica **VaciaLista(),** Retorna Cierto si la lista esta vacía **DestruyeLista():** Destruye la Lista **ConsLista(),** Retorna el elemento de la lista según una posición especifica. Si la lista esta vacía devuelve un valor especial que lo indica.

Implementación del TAD LISTA según el Método Enlazado Simple

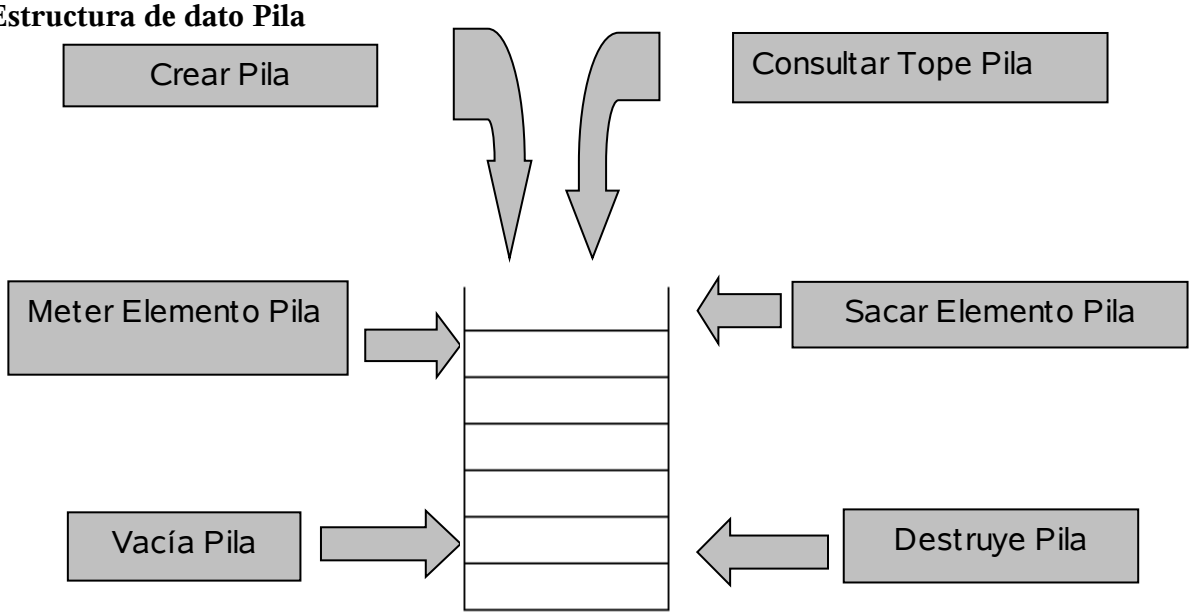
PILAS:
Son un tipo especial de lista, conocidas como listas LIFO (Last In, First Out): el último en entrar es el primero en salir). Los elementos se "amontonan" o apilan, de modo que sólo el elemento que está encima de la pila puede ser leído, y sólo pueden añadirse elementos encima de la pila.

COLAS:
Otro tipo de listas, conocidas como listas FIFO (First In, First Out: El primero en entrar es el primero en
salir). Los elementos se almacenan en fila, pero sólo pueden añadirse por un extremo y leerse por el otro.

LISTAS CIRCULARES:
También llamadas listas cerradas, son parecidas a las listas enlazadas, pero el último elemento apunta al
primero. De hecho, en las listas circulares no puede hablarse de "primero" ni de "último". Cualquier nodo puede ser el nodo de entrada y salida. Se recorren siempre en el mismo sentido.

LISTAS DOBLEMENTE ENLAZADAS:
Cada elemento dispone de dos punteros, uno apunta al siguiente elemento y el otro al elemento anterior.
Al contrario que las listas abiertas anteriores, estas listas pueden recorrerse en los dos sentidos.

Estructura de dato Pila



Especificación Pila[TipoEle]		
1	**Especificación Sintáctica** CrearPila()→Pila, InsertaElem()→Pila, EliminarElem()→Pila, ConsultaPila()→TipoEle, VaciaPila()→Logico, MostrarPila() →. DestruyePila()()→.	**CrearPila()** Crea una pila vacía **InsertaElem()** Ingresa un nuevo elemento a la pila por el tope de la misma **EliminarElem(),** Elimina el elemento que esta actualmente en el tope de la pila, si la pila no esta vacía **VaciaPila(),** Retorna Cierto si la pila esta vacía **DestruyePila():** Destruye la pila **ConsultaPila(),** Retorna el elemento apuntado por l tope de la pila. **MostrarPila()**: Imprime en pantalla todos los elementos de la pila.
2	**Declaraciones**	
3	TipoEle : e, {TipoEleNoDef} **Especificación Semántica** VaciaPila(CrearPila())=Cierto VaciaPila(InsertaElem())=Falso ConsultaPila(CrearPila())={TipoEleNoDef} EliminarElem(CrearPila())=CrearPila()	

```
/******************************
ARCHIVO DE CABECERA pila.h
******************************/
#include <iostream>
#include <cstdlib>

using namespace std;

const int  MAXI = 100;

class Pila{
  int tope;
      int PilaSec[MAXI];

      public:
      Pila(void);
      void InsertaElem(int e);
      void EliminarElem();
      int ConsultaPila();
      int VaciaPila();
      void MostrarPila();
      ~Pila();
};
Pila::Pila(){
      tope=0;
      for (int i =0;i<MAXI; i++){
            PilaSec[i] = 0;
      }
}

void Pila::InsertaElem(int e){
      if (tope == MAXI){
            cout << "\nLa Pila esta llena.";
            cin.get();
      }else
            PilaSec[tope++] = e;
}

void Pila::EliminarElem(){
      if (tope==0){
            cout << "\nLa Pila esta vacia.";
            cin.get();
      }else
            PilaSec[tope--] = 0;
}

int Pila::ConsultaPila(){
      if (tope==0){
            cout << "\nLa Pila esta vacia.";
            cin.get();
      }else
            return PilaSec[tope-1];
}

int Pila::VaciaPila(){
      return (tope == 0);
}
```

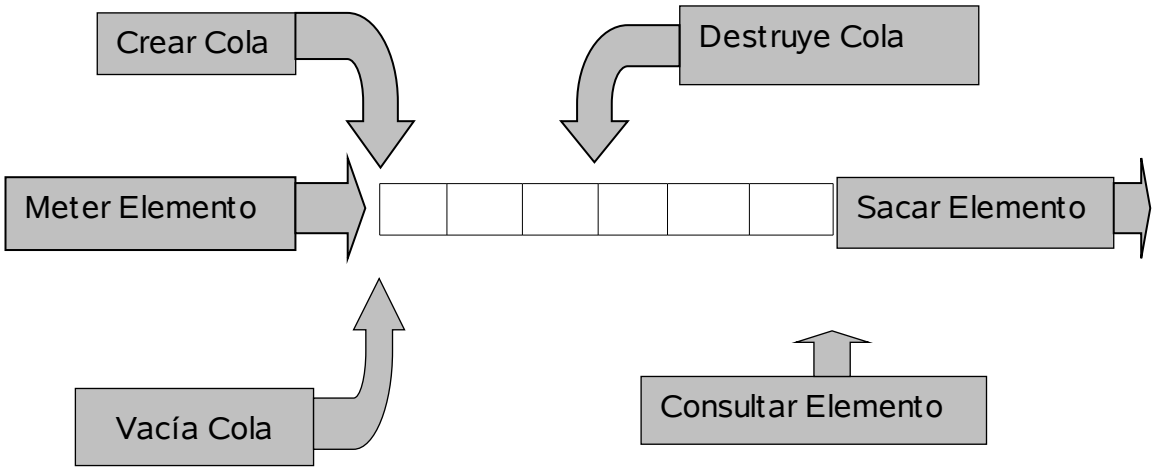
```
void Pila::MostrarPila(){
	for (int i=tope-1 ; i>=0; i--)
		cout << "Pos[" << i + 1 << "] = " << PilaSec[i] << endl;
}

Pila::~Pila(){
	tope = 0;
	for (int i =0; i<MAXI; i++)
		PilaSec[i] = 0;
}
```

//FIN DEL ARCHIVO DE CABECERA pila.h

```
/*****************************************
ARCHIVO DE PRUEBA PARA LA CLASE pila.h
pruebaPila.cpp
*****************************************/
#include <iostream>
#include <cstdlib>
#include "pila.h"

using namespace std;

int main(){
	system("clear");
	Pila miPila;

	miPila.InsertaElem(9);
		cout << "Elemento en el tope: " << miPila.ConsultaPila() << endl;
	miPila.InsertaElem(5);
	miPila.InsertaElem(3);
	miPila.InsertaElem(7);
	cout << "Elemento en el tope: " << miPila.ConsultaPila() << endl;
	miPila.MostrarPila();
	miPila.EliminarElem();
	miPila.EliminarElem();
	miPila.EliminarElem();
	cout << "Elemento en el tope: " << miPila.ConsultaPila() << endl;


	cout << "\nF I N	D E L	P R O G R A M A." << endl;
}
```

//FIN DEL ARCHIVO DE PRUEBA pruebaPila.cpp

Estructura de dato Cola



Especificación Cola[TipoEle]		
1	**Especificación Sintáctica** CrearCola()→Cola InsertaElem()→Cola EliminarElem()→Cola ConsultaCola()→TipoEle, VaciaCola()→Logico, MostrarCola() →. DestruyeCola()()→. **Declaraciones** TipoEle : e, {TipoEleNoDef} **Especificación Semántica** VaciaCola(CrearCola())=Cierto VaciaCola(InsertaElem())=Falso ConsultaCola(CrearCola())={TipoEleNo Def} EliminarElem(CrearCola())=CrearCola()	**CrearCola()** Crea una cola vacía **InsertaElem()** Ingresa un nuevo elemento a la cola por el fin de la misma **EliminarElem(),** Elimina el elemento que esta actualmente en el inicio de la cola, si la cola no esta vacía **VaciaCola(),** Retorna Cierto si la cola esta vacía **DestruyeCola():** Destruye la cola **ConsultaCola(),** Retorna el elemento apuntado por el inicio de la cola. **MostrarCola()**: Imprime en pantalla todos los elementos de la cola.
2		
3		

```
/*******************************
ARCHIVO DE CABECERA cola.h
*******************************/
#include <iostream>
#include <cstdlib>

using namespace std;

const int  MAXI = 100;

class Cola{
  int fin;
        int ColaSec[MAXI];

        public:
        Cola(void);
        void InsertaElem(int e);
        void EliminarElem();
        int ConsultaCola();
        int VaciaCola();
        void MostrarCola();
        ~Cola();
};
Cola::Cola(){
        fin=0;
        for (int i =0;i<MAXI; i++){
            ColaSec[i] = 0;
        }
}

void Cola::InsertaElem(int e){
        if (fin == MAXI){
            cout << "\nLa Cola esta llena.";
            cin.get();
        }else
            ColaSec[fin++] = e;
}

void Cola::EliminarElem(){
        if (fin==0){
            cout << "\nLa Cola esta vacia.";
            cin.get();
        }else{
            for (int i = 1; i<MAXI; i++)
                                                    ColaSec[i-1] = ColaSec[i];
                                                    fin--;
        }
}

int Cola::ConsultaCola(){
        if (fin==0){
            cout << "\nLa Cola esta vacia.";
            cin.get();
        }else
            return ColaSec[0];
}
```

```
int Cola::VaciaCola(){
	return (fin == 0);
}

void Cola::MostrarCola(){
	cout << "COLA: ";
	for (int i=0 ; i<fin; i++)
		cout << ColaSec[i] << " ";
	cout << endl;
}

Cola::~Cola(){
	fin = 0;
	for (int i =0; i<MAXI; i++)
		ColaSec[i] = 0;
}
```

//FIN DEL ARCHIVO DE CABECERA cola.h

```
/*****************************************
ARCHIVO DE PRUEBA PARA LA CLASE cola.h
pruebaCola.cpp
*****************************************/
#include <iostream>
#include <cstdlib>
#include "cola.h"

using namespace std;

int main(){
	system("clear");
	Cola miCola;

	miCola.EliminarElem();
	miCola.InsertaElem(9);
	cout << "Elemento en el Fin: " << miCola.ConsultaCola() << endl;
	miCola.InsertaElem(5);
	miCola.InsertaElem(3);
	miCola.InsertaElem(7);
	cout << "Elemento en el Fin: " << miCola.ConsultaCola() << endl;
	miCola.MostrarCola();
	miCola.EliminarElem();
	miCola.EliminarElem();
	miCola.EliminarElem();
	cout << "Elemento en el Fin: " << miCola.ConsultaCola() << endl;
	miCola.MostrarCola();

	cout << "\nF I N	D E L	P R O G R A M A." << endl;
}
```

//FIN DEL ARCHIVO DE PRUEBA pruebaCola.cpp

PROGRAMACION II. POO

Ejercicio Propuesto

1).- Usar el ejemplo pila para obtener el binario de un numero entero dado.

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

using namespace std;

int main(){
	int numero,cociente,resto;
	system("clear");
	Pila miPila, otraPila;

	cout << "Introduzca numero a transformar: ";
	cin >> numero;
	cociente = (int)(numero / 2);
	resto = numero % 2;
	miPila.InsertaElem(resto);

	while (cociente >= 1){
	miPila.InsertaElem(cociente%2);
	cociente = (int)(cociente / 2);
	}
	while (!miPila.VaciaPila()){
	otraPila.InsertaElem(miPila.ConsultaPila());
	miPila.EliminarElem();
	}
	otraPila.MostrarPila();

	cout << "\nF I N     D E L     P R O G R A M A." << endl;
}
```

2).- Usar una pila para verificar el equilibrio de paréntesis, llaves y corchetes de una ecuación, ejemplo: [a + b/(c-d)]

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include "pila.h"

using namespace std;

int main(){
        char Ecuacion[255];
        int parentesisIni, parentesisFin, corchetesIni, corchetesFin, llavesIni, llavesFin, i;



        system("clear");
        Pila miPila;

        parentesisIni = 0;
    parentesisFin = 0;
    corchetesIni = 0;
    corchetesFin = 0;
    llavesIni = 0;
    llavesFin = 0;

        cout << "Introduzca la Ecuacion: ";
        cin.getline(Ecuacion,255);

        i = 0;
        while (Ecuacion[i] != '\0'){
         miPila.InsertaElem(Ecuacion[i++]);
        }

        while (!miPila.VaciaPila()){
         switch (miPila.ConsultaPila()){
                                                                                case '(':

         parentesisIni++;
                                                                                                break;
```

```
                                                    case ')':
    parentesisFin++;
                                                    break;

                                                    case '[':
    corchetesIni++;
                                                    break;

                                                    case ']':
    corchetesFin++;
                                                    break;

                                                    case '{':
    llavesIni++;
                                                    break;

                                                    case '}':
    llavesFin++;
                                                    break;
  }
   miPila.EliminarElem();
   }
  if (parentesisIni == parentesisFin)
   cout << "Paretesis (" << parentesisIni << " abiertos) equilibrados" << endl;
  else
   cout << "Parentesis (" << parentesisFin << " cerrados) desequilibrados" << endl;

  if (corchetesIni == corchetesFin)
   cout << "Corchetes (" << corchetesIni << " abiertos) equilibrados." << endl;
  else
   cout << "Corchetes (" << corchetesFin << " cerrados) desequilibrados" << endl;
```

```
    if (llavesIni == llavesFin)
     cout << "Llaves (" << llavesIni << "abiertos) equilibradas" << endl;
    else
     cout << "Llaves(" << llavesFin << "cerrados) desequilibradas" << endl;
    cout << "\nF I N     D E L     P R O G R A M A." << endl;
}
```