Instituto Superior de Formación Técnica Nº 151
Carrera: Analista en Sistemas
Asignatura: Algoritmos y Estructuras de Datos I
Docente: Lic. José Oemig

Simulación Final B	
Modalidad: Semipresencial	**Estrategia Didáctica:** Individual
Metodología de Desarrollo: Det. docente	**Metodología de Corrección:** Det. docente
Carácter del Trabajo: Obligatorio – Con Nota	**Fecha de Entrega:** Det. docente
Alumno/a: Bravo, David Hernan	

Marco Teórico:

1. Explicar el concepto de Problema según G. Polya.

George Polya, un destacado matemático húngaro, dedicó gran parte de su carrera a estudiar los procesos mentales involucrados en la resolución de problemas. Para Polya, un problema no se limita a un ejercicio matemático; es cualquier situación que requiera encontrar una solución desconocida y para la cual no se cuenta con un método inmediato.

Un problema, en el sentido polyeano, implica una cierta tensión entre lo conocido y lo desconocido, entre lo que tenemos y lo que queremos. Es una especie de enigma que desafía nuestra capacidad de pensar y razonar. Polya destaca que la resolución de problemas no es un proceso mecánico, sino que involucra una serie de habilidades cognitivas y metacognitivas que van más allá de la simple aplicación de algoritmos.

En su obra seminal *"Cómo plantear y resolver problemas"*, Polya propone un método heurístico en cuatro fases para abordar la resolución de problemas:

a. **Comprender el problema:** En esta primera fase, es crucial identificar qué se conoce, qué se desconoce y cuáles son las condiciones del problema.
b. **Concebir un plan:** Aquí se busca establecer una conexión entre los datos conocidos y la incógnita. Puede involucrar la elección de una estrategia, la formulación de una conjetura o la búsqueda de un problema similar ya resuelto.
c. **Ejecutar el plan:** Se lleva a cabo el plan concebido, realizando los cálculos o los razonamientos necesarios para llegar a la solución.
d. **Visión retrospectiva:** Una vez encontrada la solución, se revisa el proceso completo. Se busca verificar si la solución es correcta, si existe otra forma de resolver el problema y si se pueden generalizar los resultados obtenidos.

Polya enfatiza que la resolución de problemas es una habilidad que se puede aprender y desarrollar. Al aplicar su método, los estudiantes no solo adquieren conocimientos específicos, sino que también desarrollan un pensamiento crítico y creativo que les será útil en diversas situaciones de la vida.

2. ¿Qué son los paradigmas? Relaciónelo con paradigmas de Programación.

Un paradigma, en términos generales, es un conjunto de creencias, valores, conceptos y prácticas que constituyen una visión del mundo y guían la forma en que entendemos y abordamos la realidad. En el contexto de la programación, un paradigma es un modelo conceptual que define una forma particular de pensar y estructurar los programas. Estos paradigmas ofrecen una perspectiva única sobre cómo resolver problemas computacionales, estableciendo las reglas, convenciones y herramientas que los programadores utilizan.

Los paradigmas de programación son como lentes a través de los cuales se observa y resuelve un problema. Cada paradigma enfatiza diferentes aspectos de la programación, como la gestión de datos, el control de flujo, la abstracción y la concurrencia. Algunos de los paradigmas más conocidos incluyen la programación imperativa (donde se describen los pasos a seguir de manera secuencial), la programación orientada a objetos (donde se modelan entidades del mundo real como objetos con atributos y métodos), la programación funcional (donde se enfatiza la aplicación de funciones a datos), y la programación lógica (donde se expresan problemas como conjuntos de hechos y reglas).

La elección de un paradigma de programación depende de diversos factores, como la naturaleza del problema a resolver, las herramientas disponibles y las preferencias del programador. Cada paradigma tiene sus propias fortalezas y debilidades, y a menudo los programadores combinan elementos de diferentes paradigmas para crear soluciones más eficientes y elegantes. En resumen, los paradigmas de programación proporcionan un marco conceptual fundamental para el desarrollo de software, influyendo en la forma en que los programadores diseñan, implementan y mantienen sus programas.

3. ¿Qué entiende por abstracción? Dar ejemplos de Abstracción.

La abstracción es un principio fundamental en programación que consiste en enfocarse en las características esenciales de un objeto o sistema, omitiendo los detalles de implementación que no son relevantes para el problema que se está resolviendo. Es como cuando describimos un coche: nos interesa saber que tiene ruedas, un motor y que sirve para transportarnos, pero no necesitamos conocer los detalles internos del motor o la composición exacta de los neumáticos.

En programación, la abstracción nos permite crear modelos simplificados de sistemas complejos, facilitando su comprensión, diseño y mantenimiento.

Ejemplos de Abstracción:

a. Clases y Objetos:

- **Definición de una clase:** Una clase es un plano para crear objetos. Define los atributos (datos) y los métodos (funciones) que un objeto de ese tipo tendrá. Por ejemplo:

```
class Coche
   {

      private:
         int velocidad;

      public:
         void acelerar()
            {
               // Código para acelerar el coche.
            }
         void frenar()
            {
               // Código para frenar el coche.
            }
   };
```

- **Creación de un objeto:** Un objeto es una instancia de una clase. Al crear un objeto, estamos creando una representación concreta de esa clase.

```
int main()
   {
      Coche* miCoche = new Coche();

      miCoche->acelerar();
      miCoche->frenar();

      delete miCoche;

      return 0;
   }
```

En este ejemplo, la clase Coche abstrae la complejidad de un coche real, permitiéndonos trabajar con él sin preocuparnos por detalles como el tipo de motor o el sistema de frenos.

b. Funciones:

- **Encapsulamiento de lógica:** Una función agrupa un conjunto de instrucciones para realizar una tarea específica. Por ejemplo:

```
int calcularAreaRectangulo(int base, int altura)
   {
```

```
    return base * altura;
}
```

Esta función encapsula el cálculo del área de un rectángulo, permitiendo reutilizarla en diferentes partes del programa sin tener que repetir el código.

c. Tipos de Datos Abstractos (TDA):

- **Creación de nuevos tipos de datos:** Los TDA permiten definir nuevos tipos de datos personalizados, como pilas, colas, listas, árboles, etc. Por ejemplo:

```
class Pila
{
    private:
    // Implementación interna de la pila.

    public:
        void push(int valor)
            {
                // Código para agregar un elemento a la pila.
            }
        int pop()
            {
                // Código para eliminar y devolver el último elemento.
            }
};
```

Un TDA pila oculta la implementación interna (cómo se almacenan los elementos) y expone una interfaz simple para realizar operaciones como agregar y eliminar elementos.

d. Espacios de Nombres:

- **Organización del código:** Los espacios de nombres permiten agrupar identificadores (como clases, funciones y variables) para evitar conflictos de nombres y mejorar la legibilidad del código.

```
namespace Geometria
{
    double calcularAreaCirculo(double radio)
        {
            // ...
        }
}
```

e. Abstracción de Datos:

- **Ocultar detalles de implementación:** La abstracción de datos permite ocultar los detalles internos de una estructura de datos, exponiendo solo las operaciones que se pueden realizar sobre ella. Por ejemplo, un vector puede implementarse como un arreglo dinámico, pero desde el punto de vista del

usuario solo se ofrecen las operaciones de acceso, inserción y eliminación de elementos.

4. ¿Qué entiende por Polimorfismo?

El polimorfismo, en programación orientada a objetos, es la capacidad de que un objeto pueda tomar múltiples formas. Esto significa que diferentes objetos pueden responder de manera distinta a un mismo mensaje, aunque compartan una misma interfaz. Es como si tuvieramos una caja de herramientas donde cada herramienta (objeto) tiene una función específica (mensaje) pero todas caben en una mano (interfaz).

Imaginemos tener una clase base llamada "Animal" con un método llamado "hacerSonido()". Las subclases "Perro", "Gato" y "Pato" heredan de "Animal" y sobreescriben el método "hacerSonido()" para producir sonidos distintos. Al llamar al método "hacerSonido()" en un objeto de cualquiera de estas subclases, obtendremos un resultado diferente: el perro ladrará, el gato maullará y el pato graznará.

En esencia, el polimorfismo permite que el mismo código se comporte de manera diferente en función del tipo de objeto al que se aplica. Esto hace que el código sea más flexible, reutilizable y fácil de mantener, ya que puedes tratar a objetos de diferentes clases como si fueran del mismo tipo, sin tener que preocuparse por los detalles de su implementación interna.

El polimorfismo se logra a través de:

- **Herencia:** Cuando una clase hereda de otra, adquiere los atributos y métodos de la clase base, pero puede sobreescribirlos para proporcionar una implementación más específica.
- **Interfaces:** Una interfaz define un conjunto de métodos que una clase debe implementar. Diferentes clases pueden implementar la misma interfaz de manera diferente.
- **Sobrecarga de métodos:** Permite definir múltiples métodos con el mismo nombre pero con diferentes parámetros.

5. ¿Qué es el Encapsulamiento? Dar un ejemplo en POO.

El encapsulamiento es uno de los pilares fundamentales de la programación orientada a objetos (POO). Consiste en agrupar los datos (**atributos**) y las operaciones (**métodos**) que actúan sobre esos datos dentro de una misma unidad, generalmente una clase. Esta agrupación se realiza con el objetivo de proteger los datos internos de una

clase de modificaciones no autorizadas y de proporcionar un interfaz bien definido para interactuar con esos datos.

Imaginemos una caja fuerte: los objetos de valor (datos) están almacenados en su interior y solo se puede acceder a ellos a través de una combinación específica (métodos). El encapsulamiento es como esa caja fuerte, que oculta los detalles internos de implementación y expone una interfaz pública para que otros puedan utilizarla de forma segura y controlada.

Ejemplo:

Consideremos una clase CuentaBancaria. Esta clase encapsulará información como el número de cuenta, el titular y el saldo. Para proteger estos datos de modificaciones directas, los declararemos como privados y proporcionaremos métodos públicos (getters y setters) para acceder y modificar estos valores:

```
#include <iostream>

using namespace std;

class CuentaBancaria
{
   private:
      int numeroCuenta;
      string titular;
      double saldo;

   public:
      void setNumeroCuenta(int nuevoNumero)
         {
            numeroCuenta = nuevoNumero;
         }

      int getNumeroCuenta() const
         {
            return numeroCuenta;
         }

      // ... otros getters y setters para titular y saldo.

      void depositar(double cantidad)
         {
            saldo += cantidad;
         }

      void retirar(double cantidad)
         {
            if (saldo >= cantidad)
               {
                  saldo -= cantidad;
               } else
```

```
                {
                    cout << "Fondos insuficientes" << endl;
                }
            }
};
```

En este ejemplo:

- Los atributos numeroCuenta, titular y saldo son privados, lo que significa que solo pueden ser accedidos desde dentro de la clase.
- Los métodos setNumeroCuenta, getNumeroCuenta, depositar y retirar son públicos, lo que permite a otros objetos modificar o consultar el estado de la cuenta de manera controlada.

Beneficios del encapsulamiento:

- **Protección de datos:** Evita modificaciones accidentales o malintencionadas en los datos internos de un objeto.
- **Reutilización de código:** Permite crear clases más robustas y reutilizables, ya que los detalles de implementación están ocultos.
- **Facilita el mantenimiento:** Al separar la interfaz de la implementación, es más fácil realizar cambios en el código sin afectar a otras partes del programa.
- **Abstracción:** Permite crear modelos más abstractos y cercanos al mundo real.

6. Relacionar Problema, Abstracción y Programación.

Un **problema** es cualquier situación que requiere una solución, y en el ámbito de la programación, se traduce en una necesidad de crear un programa que automatice una tarea o resuelva una cuestión específica. Para abordar este problema, el programador recurre a la **abstracción**, que consiste en simplificar la realidad, enfocándose en las características esenciales del problema y omitiendo los detalles irrelevantes. A través de la abstracción, se construyen modelos conceptuales que representan los elementos del problema y sus relaciones.

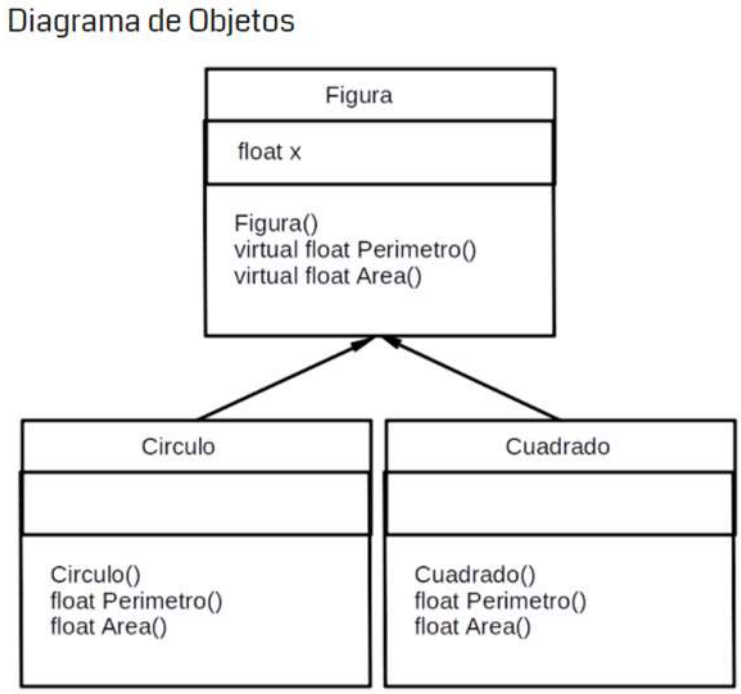
La **programación** es el proceso de transformar estos modelos abstractos en código ejecutable. Los lenguajes de programación proporcionan las herramientas necesarias para crear estructuras de datos, algoritmos y funciones que implementan las soluciones a los problemas. La abstracción guía este proceso, permitiendo al programador descomponer un problema complejo en subproblemas más manejables y crear soluciones modulares y reutilizables.

Marco Práctico:

Instituto Superior de Formación Técnica Nº 151
Carrera: Analista en Sistemas
Asignatura: Algoritmos y Estructuras de Datos I
Docente: Lic. José Oemig

Se pide desarrollar una solución que Implemente el diagrama de clases adjunto, con las siguientes condiciones:

1. Implementar tanto el .hpp como el .cpp de las clases.
2. Realizar la Herencia correspondiente.
3. Implementado Funciones Virtuales Puras (La Clase Figura debe ser Abstracta).
4. En el Main deberá modularizar el desarrollo respetando las pautas tratadas en Clase.
5. La Aplicación Deberá solicitar 3 figuras que deberán ser tratadas Polimórficamente.
6. Deberá Crear Puntero de array de 3 elementos en memoria Dinámica para manejar las Figuras y tratarlas todas como su clase base.
7. Implementando Polimorfismo recorrer las figuras mostrando el perímetro y el área de cada una.
8. El programa deberá compilar y correr sin errores, cumpliendo el 100% con lo solicitado.

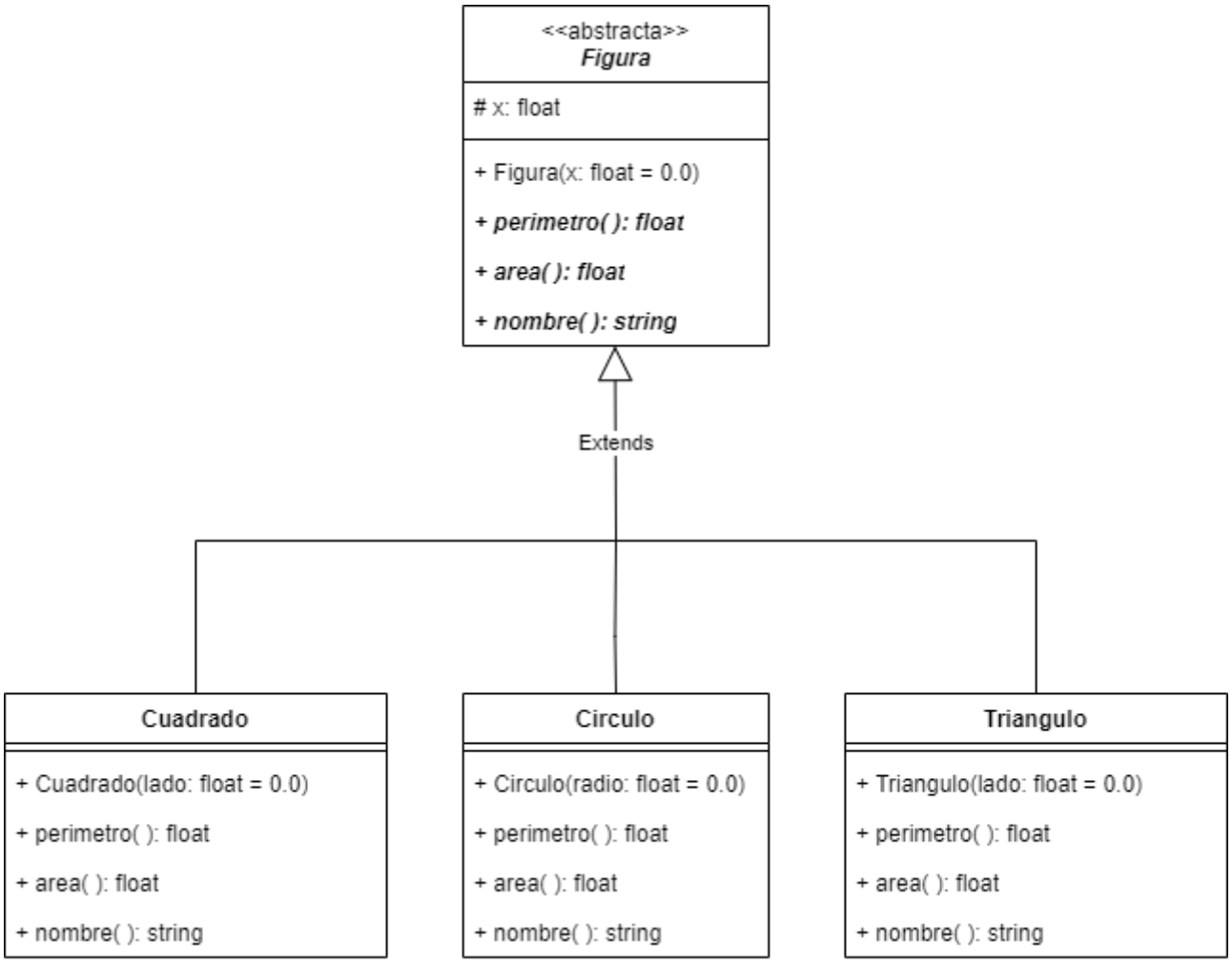


Nota:

- Perímetro de un círculo = 2. π x r
- El área del círculo es igual al producto de π por el radio (r) al cuadrado.
- Perímetro del Cuadrado 4 * Lado
- Área del Cuadrado Lado x Lado

DIAGRAMA DE CLASES

Instituto Superior de Formación Técnica Nº 151
Carrera: Analista en Sistemas
Asignatura: Algoritmos y Estructuras de Datos I
Docente: Lic. José Oemig



EXPLICACIÓN DE LAS RELACIONES ENTRE CLASES

1. Figura (Clase Base Abstracta):

   - Es una clase abstracta que no puede ser instanciada directamente.
   - Contiene atributos y métodos comunes para todas las figuras, incluyendo el atributo protegido x (utilizado para parámetros como radio, lado, etc.).
   - Define tres métodos virtuales puros (Perimetro, Area, y Nombre), obligando a las clases derivadas a implementar estas funciones.

2. Relación de Herencia:

   - Las clases Circulo, Cuadrado, y Triangulo heredan de Figura.
   - La relación es de herencia simple, representada como Clase Derivada --> Clase Base.
   - Las clases derivadas sobrescriben los métodos virtuales puros para proporcionar implementaciones específicas según su geometría.

3. Polimorfismo:

   - Todas las clases (Circulo, Cuadrado, y Triangulo) se pueden tratar como objetos de tipo Figura debido a la herencia y al uso de punteros polimórficos.

- Esto permite almacenar diferentes figuras en un array dinámico y llamarlas usando el mismo interfaz (Perimetro, Area, Nombre).

4. No existe composición o agregación:

- Ninguna clase tiene instancias de otras como atributos.
- Cada clase es autónoma y no depende de la composición o agregación.

MAIN.CPP

```
#include "include/circulo.hpp"
#include "include/cuadrado.hpp"
#include "include/triangulo.hpp"

#include <iostream>

using namespace std;

int main()
{
   // Crear puntero de array dinámico de 3 elementos.
   Figura* figuras[3];

   // Solicitar datos para cada figura.
   figuras[0] = new Circulo(5.5);   // Radio = 5.5.
   figuras[1] = new Cuadrado(4.3); // Lado = 4.3.
   figuras[2] = new Triangulo(3.1); // Lado = 3.1.

   cout << "FORMITAS GEOMETRICAS" << endl;
   cout << "********************" << endl;

   // Recorrer las figuras y mostrar perímetro y área.
   for (int i = 0; i < 3; ++i)
      {
         cout << "Figura " << i + 1 << ": " << figuras[i]->nombre() << endl;
         cout << "Perimetro: " << figuras[i]->perimetro() << endl;
         cout << "Area: " << figuras[i]->area() << endl << endl;
      }

   cout << "Mmm... Papitas Fritas Geometricas... (icono de baba va aqui)" << endl;

   // Libera la memoria.
   for (int i = 0; i < 3; ++i)
   {
      delete figuras[i];
   }

   return 0;
}
```

INCLUDE/CIRCULO.HPP

```
#ifndef CIRCULO_HPP
#define CIRCULO_HPP

#include "figura.hpp"

class Circulo : public Figura
   {
```

```
    public:
        Circulo(float radio = 0.0);
        virtual ~Circulo() {}

        float perimetro() const override;
        float area() const override;
        string nombre() const override;
};

#endif
```

SRC/CIRCULO.CPP

```
#include "../include/circulo.hpp"

Circulo::Circulo(float radio)
    : Figura(radio) {}

float Circulo::perimetro() const
    {
        return 2 * 3.14159265359 * x; // Aproximación de PI para el cálculo del perímetro.
    }

float Circulo::area() const
    {
        return 3.14159265359 * x * x; // Área del círculo es PI * radio².
    }

string Circulo::nombre() const
{
    return "Circulo";
}
```

INCLUDE/CUADRADO.HPP

```
#ifndef CUADRADO_HPP
#define CUADRADO_HPP

#include "figura.hpp"

class Cuadrado : public Figura
    {
        public:
            Cuadrado(float lado = 0.0);
            virtual ~Cuadrado() {}

            float perimetro() const override;
            float area() const override;
            string nombre() const override;
    };

#endif
```

SRC/CUADRADO.CPP

```
#include "../include/cuadrado.hpp"
```

```
Cuadrado::Cuadrado(float lado)
    : Figura(lado) {}

float Cuadrado::perimetro() const
    {
        return 4 * x; // Perímetro de un cuadrado.
    }

float Cuadrado::area() const
    {
        return x * x; // Área del cuadrado es lado².
    }

string Cuadrado::nombre() const
    {
        return "Cuadrado";
}
```

INCLUDE/TRIANGULO.HPP

```
#ifndef TRIANGULO_HPP
#define TRIANGULO_HPP

#include "figura.hpp"

class Triangulo : public Figura
    {
        public:
            Triangulo(float lado = 0.0);
            virtual ~Triangulo() {}

            float perimetro() const override;
            float area() const override;
            string nombre() const override;
    };

#endif
```

SRC/TRIANGULO.CPP

```
#include "../include/triangulo.hpp"

Triangulo::Triangulo(float lado)
    : Figura(lado) {}

float Triangulo::perimetro() const
    {
        return 3 * x; // Perímetro de un triángulo equilátero.
    }

float Triangulo::area() const
    {
        float altura = (0.86602540378) * x; // 0.86602540378 es el cuadrado de (3)/2.
        return (x * altura) / 2; // Área del triángulo equilátero.
    }
```

```
string Triangulo::nombre() const
   {
      return "Triangulo";
}
```

```
FORMITAS GEOMETRICAS
**********
Figura 1: Circulo
Perimetro: 34.5575
Area: 95.0332

Figura 2: Cuadrado
Perimetro: 17.2
Area: 18.49

Figura 3: Triangulo
Perimetro: 9.3
Area: 4.16125

Mmm ...  Papitas Fritas Geometricas ... (icono de baba va aqui)
Presione una tecla para continuar . . .
```

Instituto Superior de Formación Técnica Nº 151
Carrera: Analista en Sistemas
Asignatura: Algoritmos y Estructuras de Datos I
Docente: Lic. José Oemig